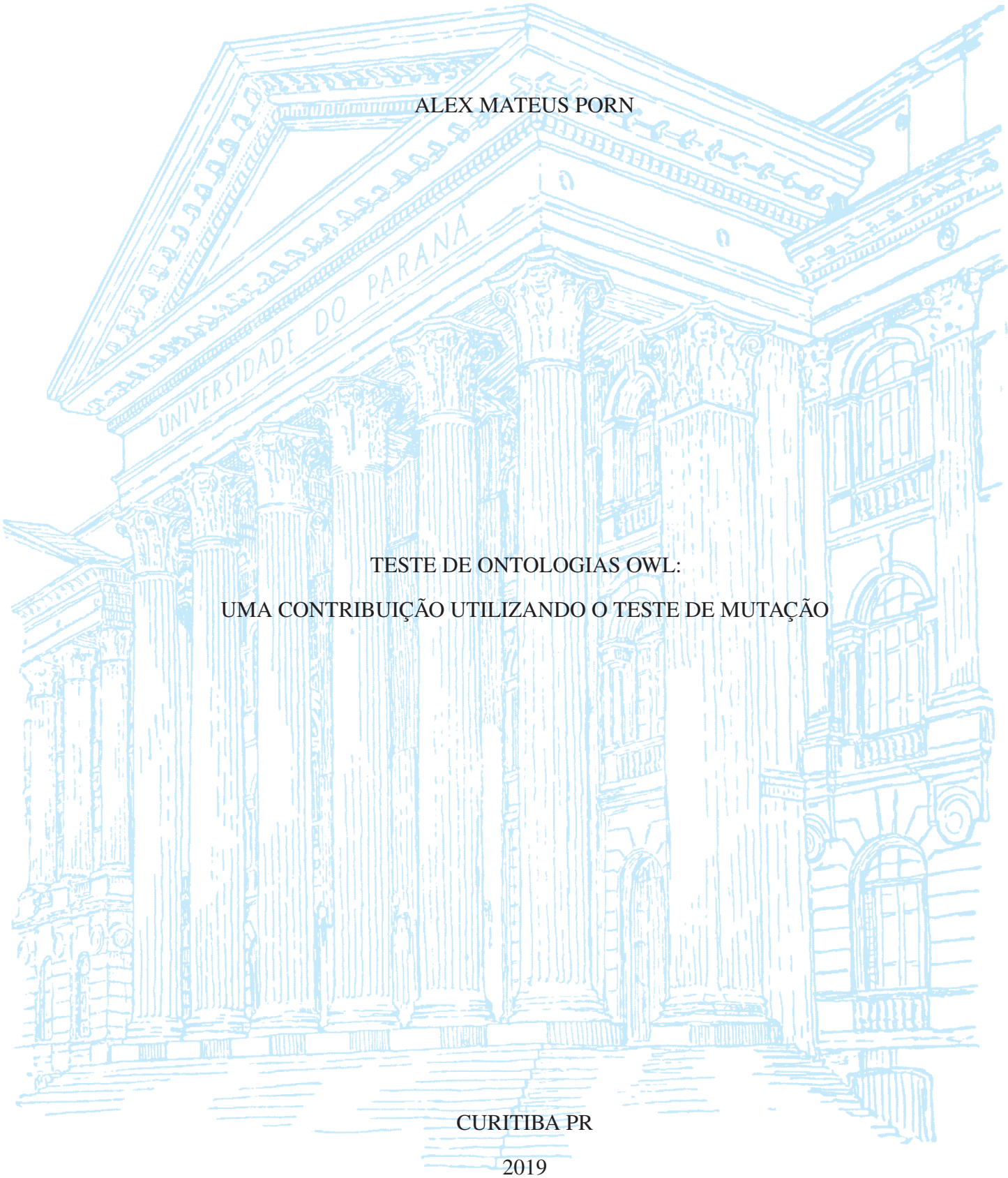
UNIVERSIDADE FEDERAL DO PARANÁ

ALEX MATEUS PORN



TESTE DE ONTOLOGIAS OWL:
UMA CONTRIBUIÇÃO UTILIZANDO O TESTE DE MUTAÇÃO

CURITIBA PR

2019

ALEX MATEUS PORN

TESTE DE ONTOLOGIAS OWL:
UMA CONTRIBUIÇÃO UTILIZANDO O TESTE DE MUTAÇÃO

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Profª. Drª. Leticia Mara Peres.

CURITIBA PR

2019

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

P836t Porn, Alex Mateus
Teste de ontologias OWL: uma contribuição utilizando o teste de mutação [recurso eletrônico] / Alex Mateus Porn – Curitiba, 2019.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.
Orientadora: Profª. Drª. Letícia Mara Peres.

1. Ontologia. 2. Modelagem de dados. 3. Teste de mutação. I. Universidade Federal do Paraná. II. Peres, Letícia Mara. III. Título.

CDD: 006.35

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA
40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **ALEX MATEUS PORN** intitulada: **TESTE DE ONTOLOGIAS OWL: UMA CONTRIBUIÇÃO UTILIZANDO O TESTE DE MUTAÇÃO**, sob orientação da Profa. Dra. LETICIA MARA PERES, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua Aprovação no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 01 de Outubro de 2019.

LETICIA MARA PERES
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

MARCOS DIDONET DEL FABRO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

JORGE AUGUSTO MEIRA
Avaliador Externo (UNIVERSITé DU LUXEMBURG)

RITA CRISTINA GALARRAGA BERARDI
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

Rua Cel. Francisco H. dos Santos, 100 - Centro Politécnico da UFPR - CURITIBA - Paraná - Brasil
CEP 81531-980 - Tel: (41) 3361-3101 - E-mail: ppginf@inf.ufpr.br

”Se eu vi mais longe, foi por estar de pé sobre ombros de gigantes.“

Isaac Newton

AGRADECIMENTOS

Antes de tudo, agradeço a Deus por me guiar, iluminar e me dar tranquilidade para seguir em frente com os meus objetivos e não desanimar com as dificuldades.

Agradeço a minha esposa Cibelli Cristina Porn, pelo companheirismo durante todo o período em que estive ausente para concluir o Doutorado, onde muitas vezes estando presente, me fazia ausente para cumprir com as atividades. Com brilho nos olhos quero também agradecer ao maior presente que ela poderia ter me dado durante esse período, minha filha Emília Luísa Porn, que chegou como uma inspiração em meu último ano de curso, proporsionando um incentivo ainda maior para atingir esse objetivo. Também não posso deixar de agradecer a minha sogra Eugênica Kravec, que me auxíliou durante os primeiros meses de vida da Emília, disponibilizando seu tempo para cuidar dela enquanto eu me dedicava para concluir esta tese.

Quero agradecer aos meus pais Lucas Porn e Edith Luisa Schmidt Porn, pelo reconhecimento e orgulho ao saberem do esforço e dedicação para a obtenção desse título.

Agradeço também a Profª. Drª. Letícia Mara Peres, por toda a sua dedicação em nossas reuniões de orientação, que pela qualidade de seu trabalho foram mais do que reuniões, foram verdadeiras aulas.

Agradeço ao Prof. Dr. Marcos Didonet Del Fabro, que contribuiu positivamente com seus conselhos e contribuições, enriquecendo este trabalho.

Agradeço aos demais professores da Banca Examinadora, Prof. Dr. Jorge Augusto Meira e Profª. Drª. Rita Cristina Galarraga Berardi, pela disponibilidade em avaliar este trabalho e pelas relevantes considerações apresentadas.

Quero também agradecer a minha colega de Doutorado, Cristiane Aparecida Gonçalves Huve, pela participação e companheirismo em diversos estudos que culminaram em ótimas publicações. Agradeço também aos colegas do grupo de pesquisa, Luis Felipe Lima e Lucilia Yoshie Araki, pelas contribuições durante nossas reuniões do grupo, e também pelos momentos de descontração.

Agradeço a Profª M.ª Edna Satiko Eiri Trebien, que é a precursora de toda essa minha caminhada entre Graduação, Mestrado e Doutorado.

Agradeço ao Instituto Federal do Paraná - IFPR, representado por sua reitoria, em especial ao Câmpus União da Vitória, representado pela Direção Geral, pela oportunidade concedida para a obtenção deste título.

Agradeço também a todos os professores da área de informática do IFPR Câmpus União da Vitória, que se mobilizaram para lecionar minhas aulas durante o período em que esitve afastado.

Quero também agracer a Universidade Federal do Paraná - UFPR, especialmente ao Programa de Pós-Graduação em Informática - PPGInf, por proporcionar um curso de Doutorado com altíssima qualidade reconhecida.

Agradeço também a todos que se fizeram presentes, direta ou indiretamente, e contribuíram para a conclusão desta pesquisa.

RESUMO

Ontologias são compostas por restrições e axiomas que possibilitam descrever formalmente os conceitos, relacionamentos e propriedades de um domínio. A descrição de um conceito pode ter diferentes significados e representações através da definição de axiomas. Essa variedade de representações permite a ocorrência de defeitos durante o desenvolvimento de ontologias, que podem ocasionar falhas ou erros inesperados. Outros fatores estão relacionados ao desconhecimento pelo desenvolvedor do uso dos operadores lógicos, a enganos em decisões de modelagem, a inferências lógicas e a definições dos requisitos da ontologia. Os defeitos também podem ser inseridos pelo desenvolvedor que mesmo tendo conhecimento sobre métodos e técnicas de modelagem e construção de ontologias, pode não conhecer padrões de defeitos que possam ser ocasionados. Com o objetivo de diminuir o número de defeitos na modelagem de ontologias, é apresentada nesta tese uma contribuição ao teste de ontologias utilizando o teste de mutação. Para auxiliar na identificação dos defeitos que podem ser ocasionados, foi elaborada uma lista contendo os principais defeitos cometidos ao desenvolver ontologias. Estes defeitos foram utilizados como base para o desenvolvimento de um conjunto de 38 operadores de mutação. Para a aplicação do teste de mutação, é apresentado um método para a geração do conjunto de dados de teste, que foi automatizado com a implementação da ferramenta MutaOnto, assim como a automatização da geração dos mutantes. Para avaliar o conjunto de operadores de mutação, o método de geração dos dados de teste e a aplicação do teste de mutação para ontologias, foram realizados dois experimentos. O primeiro experimento foi realizado com ontologias construídas por desenvolvedores inexperientes, com ontologias que poderiam conter defeitos mais simples e mais fáceis de serem identificados. O segundo experimento foi realizado com ontologias selecionadas de uma base *online*, construídas por especialistas do domínio e do desenvolvimento de ontologias. Dada a experiência desses especialistas, essas ontologias não poderiam conter defeitos, ou seus defeitos deveriam ser mais difíceis de serem revelados. Para auxiliar o processo de aplicação dos dados de teste e análise dos resultados foi utilizada a ferramenta *Protégé*. O método de teste possibilitou revelar 16 tipos de defeitos identificados pelas mutações realizadas por 19 operadores de mutação, sendo que 2 tipos de defeitos foram descobertos após a aplicação do teste, pois não constam no levantamento realizado para a elaboração dos operadores. Quanto aos operadores de mutação que não revelaram algum tipo de defeito, permitiram analisar a qualidade dos dados de teste em identificar as mutações e indicar a ausência do defeito simulado por estes operadores.

Palavras-chave: Ontologias, Teste de mutação, Dados de teste

ABSTRACT

Ontologies are composed of constraints and axioms that make it possible to formally describe the concepts, relationships, and properties of a domain. The description of a concept can have different meanings and representations by defining axioms. This variety of representations enable the occurrence of defects during the development of ontologies that can cause unexpected failures or errors. Other factors are related to the unfamiliarity by the developer of the use of logical operators, mistakes in modelling decisions, logical inferences and definitions of ontology requirements. Defects can also be inserted by the developer, who, although having knowledge about ontology modelling and construction methods and techniques, may not know the patterns of defects that may be caused. In order to reduce the number of defects in ontology modelling, a contribution to the ontology test using the mutation test is presented in this thesis. To help in identifying the defects that may be caused, a list was drawn up containing the main defects committed when developing ontologies. These defects were used as the basis for the development of a set of 38 mutation operators. For the application of the mutation test, a method is presented for the generation of the test data set, which was automated with the implementation of the MutaOnto tool, as well as the automation of the mutant generation. In order to evaluate the set of mutation operators, the method of generating the test data and the application of the mutation test for ontologies, two experiments were performed. The first experiment was carried out with ontologies built by inexperienced developers, with ontologies that could contain simpler and easier to identify defects. The second experiment was carried out with ontologies selected from an online base, built by experts in the domain and ontology development. Given the experience of these experts, these ontologies could not contain defects, or their defects should be more difficult to disclose. The Protégé tool was used to assist in the application of test data and results analysis. The test method made it possible to reveal 16 types of defects identified by the mutations performed by 19 mutation operators, and 2 types of defects were discovered after the test, since they are not included in the survey carried out for the elaboration of the operators. As for the mutation operators that did not reveal any type of defect, they allowed to analyse the quality of the test data in identifying the mutations and to indicate the absence of the simulated defect by these operators.

Keywords: Ontologies, Mutation test, Test data

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

CQ	*Competency Question*
CT	Conjunto de Casos de Teste
CWA	*Closed-World Assumption*
DAML+OIL	*DARPA Agent Markup Language plus Ontology Inference Layer*
DL	*Description Logic*
ODM	*Ontology Definition Metamodel*
OLM	Conjunto de Operadores Lógicos de Mutação
openEHR	*open Eletronic Health Record*
OM	Conjunto de Operadores de Mutação
OME	Conjunto de Operadores de Mutação Estrutural
OWA	*Open-World Assumption*
OWL	*Ontology Web Language*
OWL-S	*Ontology Web Language for Services*
RDFS	*Resource Description Framework Schema*
SPARQL	*Protocol and RDF Query Language*
XML	*eXtensible Markup Language*
W3C	*World Wide Web Consortium*

LISTA DE SÍMBOLOS

$\mathcal{ALC}$	Linguagem de lógica descritiva
$\mathcal{E}$	Linguagem de lógica descritiva
$\mathcal{H}$	Linguagem de lógica descritiva
$\mathcal{I}$	Linguagem de lógica descritiva
$\mathcal{N}$	Linguagem de lógica descritiva
$\mathcal{O}$	Linguagem de lógica descritiva
$\mathcal{S}$	Abreviação para $\mathcal{ALC}$
$\mathcal{SHOIN}(\mathcal{D})$	Linguagem de lógica descritiva expressiva
$\mathcal{U}$	Linguagem de lógica descritiva
Ω	Conjunto de instâncias de uma ontologia

SUMÁRIO

1 INTRODUÇÃO

As ontologias são modelos de domínio para representação do conhecimento, sendo utilizadas na computação para a especificação de sistemas em diversas áreas. Assim como em qualquer software, elas podem conter defeitos na representação de seus conceitos e necessitam ser avaliadas e testadas dentro do contexto para o qual foram projetadas. Desse modo, métodos de teste para identificação de defeitos, tanto no desenvolvimento quanto na seleção de ontologias existentes tornam-se necessários para garantir a correta representação do domínio do conhecimento abordado. Devido ao uso generalizado de ontologias na computação, tem-se fundamentado a realização dessa pesquisa no processo de avaliação e teste de ontologias, identificando defeitos inseridos durante a fase de desenvolvimento. Desse modo, é apresentada na Seção 1.1 uma visão geral dos conceitos e representação de ontologias. Na Seção 1.2 é descrita a motivação que incitou o desenvolvimento desse trabalho. Na Seção 1.3 é apresentado o objetivo geral e os objetivos específicos para essa finalidade. As contribuições desta tese são apresentadas na Seção 1.4 e na Seção 1.5 é apresentada a organização do trabalho.

1.1 VISÃO GERAL

Ontologias são representações formais de conceitos e relacionamentos que permitem facilitar o compartilhamento e reúso de informações (Davies et al., 2003). Conforme Gruber (1995) e Guarino (1995), na computação uma ontologia refere-se a um termo técnico para representar um artefato que é projetado para a finalidade de permitir a modelagem de conhecimento sobre algum domínio real ou imaginário.

Uma ontologia é definida com base em um conjunto de conceitos com os quais modela um domínio. Esses conceitos são tipicamente classes, propriedades e relacionamentos entre os membros das classes. As definições desses conceitos incluem informações sobre os significados e as restrições sobre sua coerência na lógica aplicada (Gruber, 1995). Diante desse modo de representação de conceitos, as ontologias demonstraram ser úteis para apoiar a especificação e desenvolvimento de sistemas de computação. Outras características do modo de implementação que favorecem o seu uso na computação é o fato de fornecerem uma descrição exata de um conceito e um vocabulário para representação do conhecimento, possibilitarem o mapeamento da linguagem da ontologia, estenderem o uso de uma ontologia genérica para um domínio específico e, permitirem o compartilhamento desse conhecimento (Guarino, 1995).

A representação e descrição de diversos tipos de domínio proporcionou o uso generalizado das ontologias na computação, conforme já abordado em diversas áreas, como em geoprocessamento (Sousa e Leite, 2005), educação (Prado, 2005), saúde (Nardon, 2003) e interoperabilidade de padrões de desenvolvimento de sistemas na área médica (Porn et al., 2015), destacando-se como uma possibilidade flexível de representação de informações clínicas. Um dos fatores que proporciona essa representatividade é devido ao modo de implementação das ontologias ser baseado na Suposição de Mundo Aberto - OWA (do inglês, *Open-World Assumption*), que é uma representação usada na definição do conhecimento para codificar a noção informal de que nenhum agente tem o conhecimento completo, o que permite usar a negação como insatisfatória, ou seja, dizer que algo é falso apenas se puder ser provado (Sequeda, 2015). Conforme Sequeda (2015), esse modo de representação permite que a partir de informações incompletas seja possível a descoberta de novos conhecimentos. Por exemplo, ao representar o histórico clínico de um paciente e não incluir uma alergia específica, seria incorreto afirmar que

o paciente não possui essa alergia. Ao contrário de sistemas baseados em Suposição de Mundo Fechado - CWA (do inglês, *Closed-World Assumption*), onde o sistema retornaria que o paciente não sofre dessa alergia.

Com o advento da *Web* semântica (Berners-Lee et al., 2001) as ontologias passaram a desempenhar um papel fundamental com o objetivo de tornar a informação processável para os computadores e compreensível para os humanos, devido à sua capacidade taxonômica e a possibilidade da inclusão de regras de inferência baseadas na definição de restrições e axiomas para um domínio (Berners-Lee et al., 2001; Gruber, 1995). Conforme exposto por Hendler (2001), ao contrário do que é proposto nas áreas de inteligência artificial e engenharia do conhecimento, que se concentram na criação de ontologias genéricas, a *Web* semântica baseia-se em várias ontologias pequenas e altamente contextualizadas, desenvolvidas localmente por engenheiros de software e não especialistas em ontologias.

Diversos fatores como a variedade de representações de domínio e o desenvolvimento por profissionais não especialistas, podem provocar a ocorrência de defeitos no desenvolvimento de ontologias, que frequentemente representam uma falha ou poderiam levar a ontologia a erros inesperados (Poveda-Villalón et al., 2014). Esses defeitos em sua maioria estão relacionados ao desconhecimento pelo desenvolvedor da ontologia com relação à suposição de mundo aberto, o que proporciona a utilização incorreta de operadores lógicos, enganos em decisões de modelagem, inferências lógicas e definição dos requisitos da ontologia, impactando na sua usabilidade (Poveda-Villalón et al., 2010; Rector et al., 2004). Em outros casos, defeitos podem ser inseridos pelo desenvolvedor que mesmo conhecendo as técnicas de modelagem de ontologias, desconhece padrões de defeitos que podem ser cometidos (Poveda-Villalón et al., 2010).

Conforme Gómez-Pérez (2004), Rector et al. (2004), Poveda-Villalón et al. (2010) e Corcho et al. (2009), a existência de defeitos em ontologias pode causar falhas como a instanciação incorreta de classes e objetos, retornando resultados incompletos ou incorretos, podendo invalidar uma ontologia inteira. Esses defeitos podem não ser detectados durante a execução de uma consulta na ontologia e geralmente são cometidos na definição e representação de axiomas, que são definições utilizadas pelo desenvolvedor para representar os conceitos do domínio na ontologia. Assim, as ontologias necessitam ser testadas dentro do contexto de conhecimento ao qual se propõem a descrever, do mesmo modo como um desenvolvedor de software promove os testes necessários para que o programa atenda às necessidades ao qual ele foi criado.

Métodos de testes já foram propostos para avaliar, verificar, validar ou atender critérios de qualidade das ontologias. Esses métodos analisam a instanciação de classes ou verificam inconsistências ao adicionar novos objetos, ambos realizados durante a execução das ontologias com o auxílio de um classificador (do inglês, *reasoner*) (García-Ramos et al., 2009; Vrandečić e Gangemi, 2006). A avaliação é a tarefa de medir a qualidade de uma ontologia. A qualidade pode estar associada a diferentes cenários, como identificação de erros e omissões que possam causar a incapacidade de uso de uma ontologia. Conforme Vrandečić (2010), critérios de qualidade de ontologias correspondem a características de qualidade feitas como um acordo durante a conceitualização ontológica, para tentar refletir a correta representação das definições do mundo real, a capacidade de reutilizar e compartilhar o conhecimento, o entendimento da clareza ou sua integridade. Em um trabalho preliminar (Porn et al., 2016), destacamos que esses métodos e técnicas de testes e avaliação de ontologias possibilitam analisar os resultados obtidos após a execução de uma ontologia, mas poucos métodos ou técnicas que permitam revelar quais são os defeitos existentes são explorados na literatura.

Com a finalidade de revelar defeitos, o teste de mutação demonstrou seu valor para avaliar sistemas e seus conjuntos de testes em vários domínios (Jia e Harman, 2011). Mesmo apresentando uma complexidade de mutação na ordem $O(n^2)$, onde n é a quantidade de referências

de variáveis em um programa (Wong et al., 1995), mostrou empiricamente ser eficaz na detecção de defeitos (DeMillo et al., 1978; Budd, 1980). Assim, com o objetivo de identificar os defeitos existentes em ontologias OWL (do inglês, *Ontology Web Language*), nessa pesquisa é aplicado o teste de mutação para ontologias OWL com a finalidade de identificar defeitos na definição estrutural e em axiomas lógicos de ontologias OWL.

Também é apresentado um método para a geração dos dados de teste para a execução do teste de mutação. Conforme DeMillo e Offut (1991), se por um lado é possível que para qualquer programa exista sempre um conjunto de dados de teste finito e confiável, por outro lado não existe um procedimento efetivo para gerar dados de teste confiáveis ou avaliar se o conjunto de dados de teste é ou não confiável. O domínio de entrada de uma ontologia *O*, denotado por *D(O)*, é o conjunto de todos os valores possíveis que podem ser utilizados em *O*, o que pode tornar esta abordagem impraticável dependendo da cardinalidade de *D(O)*. Assim, conforme proposto para o teste de mutação para programas (Delamaro et al., 2007), é necessário encontrar maneiras de utilizar apenas um subconjunto reduzido de *D(O)* para testar a ontologia. Nesse contexto, o método de geração dos dados de teste proposto produz questões de consultas para ontologias, geradas pelas mutações realizadas pelos operadores de mutação nas definições lógicas da ontologia, que estejam definidas em lógica descritiva e atendam a um conjunto mínimo de operações lógicas. Deste modo, cada um dos operadores de mutação que geram os dados de teste determina os subconjuntos de dados de teste.

Desse modo, considerando os defeitos que podem ser inseridos durante o desenvolvimento de ontologias OWL e a relevância da aplicação das ontologias na computação, essa pesquisa tem como um de seus objetivos a definição e estudo da utilização de operadores de mutação para axiomas lógicos. Esses operadores em conjunto com os operadores de mutação que propomos em um trabalho anterior (Porn, 2014), abrangem tanto a representação estrutural como a conceituação lógica das ontologias OWL. Tanto a geração dos mutantes quanto a geração dos dados de teste foi automatizada com o desenvolvimento de uma ferramenta para essas atividades.

1.2 MOTIVAÇÃO

Conforme abordado na literatura (Corcho et al., 2009; Gómez-Pérez, 2004; Poveda-Villalón et al., 2010; Rector et al., 2004) diversos tipos de defeitos podem ser ocasionados pelo desenvolvedor ao modelar uma ontologia. Esses defeitos podem causar uma simples falha ou até invalidar uma ontologia inteira (Poveda-Villalón et al., 2014). Com a finalidade de revelar defeitos durante o desenvolvimento de ontologias, em Porn (2014) e Porn e Peres (2014) adaptamos o teste de mutação para testar ontologias, onde propomos 25 operadores de mutação baseados na linguagem OWL. Esse método de teste foi aplicado em um estudo de caso onde modelos clínicos representados por arquétipos *openEHR* foram convertidos para OWL e testados. Após a aplicação do teste de mutação nos modelos OWL, foi possível identificar 187 ocorrências de erros em um total de 2000 ontologias mutantes geradas manualmente. Como os operadores de mutação apresentados foram definidos somente com base na linguagem OWL, não sendo considerados os modos como os defeitos são ocasionados, não foi possível identificar a quais defeitos essas 187 ocorrências de erros se referem. Assim, essa quantidade de ocorrência de erros não significa que sejam 187 defeitos diferentes, podendo um mesmo defeito estar associado a várias ocorrências de erros, implicando que na correção de um defeito, outros defeitos sejam corrigidos simultaneamente. Por outro lado, como as ontologias testadas foram obtidas a partir da transformação de modelos de dados clínicos, os defeitos ocasionados podem ter sido gerados devido à existência de defeitos nesses modelos clínicos ou gerados pelo próprio processo de transformação, não sendo deste modo, defeitos exclusivos de ontologias OWL.

Em Porn (2014) não apresentamos um método padrão de geração de dados de teste, sendo esses dados formulados em lógica descritiva de modo manual e aleatório pelo testador. Esse método de geração dos dados de teste possibilita que o testador cometa equívocos, como a geração de um conjunto de dados de teste incompleto, pelo desconhecimento de todos os conceitos do domínio representado na ontologia. Essa situação pode se agravar quando são testadas ontologias cada vez maiores, que abordem domínios e subdmínios, com centenas ou milhares de representações de conceitos.

Com base nos resultados desses dois estudos (Porn, 2014; Porn e Peres, 2014), o cenário motivacional dessa pesquisa é o teste de mutação para ontologias OWL que aborde tanto as mutações em representações estruturais quanto em definições de axiomas lógico-descritivos. A definição de operadores que realizem essas mutações possibilita que todas as variáveis da linguagem OWL sejam compreendidas e, que os possíveis defeitos para ontologias apresentados na literatura sejam associados aos operadores de mutação, possibilitando identificar o tipo do defeito quando esse for revelado. Essa identificação proporcionará maior confiabilidade no uso das ontologias, independente da sua área de aplicação. Neste mesmo cenário, motiva-se propor um método para a geração dos dados de teste, que aborde todas as definições conceituais representadas na ontologia através de axiomas, diminuindo consequentemente o problema da geração de um conjunto de dados de teste incompleto devido o desconhecimento do testador de todos os conceitos representados na ontologia. Por outro lado, a definição de um método padronizado para a geração dos dados de teste possibilitará garantir que todos os conceitos representados na ontologia serão testados. Diante disso, percebe-se também a importância na implementação de uma ferramenta para gerar automaticamente esses dados de teste e os mutantes, com base nos operadores de mutação. A automatização dessa atividade proporcionará agilizar a execução do teste de mutação e evitar possíveis situações como a geração de mutantes duplicados ou que variáveis da linguagem OWL não sejam mutadas por falha do testador, devido à realização manual dessa atividade.

1.3 OBJETIVOS

Diante das possibilidades de representações de domínios de conhecimento com a implementação de ontologias OWL, e da diversidade de defeitos que podem ser inseridos durante o seu desenvolvimento, propõem-se os seguintes objetivos:

- **Objetivo geral**
  - Propor uma abordagem de teste para ontologias OWL com o teste de mutação para revelar defeitos ocasionados durante a criação de ontologias, auxiliando no processo de avaliação.
- **Objetivos específicos**
  - Explorar o teste de mutação para o teste de axiomas OWL;
  - Apresentar um catálogo de possíveis defeitos ocasionados em ontologias para auxiliar na definição dos operadores de mutação.
  - Propor operadores de mutação compatíveis com a sintaxe da linguagem OWL para gerar mutantes;
  - Automatizar a geração dos mutantes e a geração dos dados de teste a partir dos axiomas OWL;
  - Avaliar a abordagem do teste de mutação para ontologias OWL.

1.4 CONTRIBUIÇÃO

Esta tese contribui no processo de avaliação de ontologias auxiliando em diversos aspectos na definição e aplicação do teste de mutação para ontologias OWL, baseando-se em alguns conceitos da proposta apresentada por Porn (2014). A contribuição desta tese abrange: (i) definição dos termos engano, defeito, erro e falha no contexto de ontologias para o teste de mutação; (ii) caracterização de dados e casos de teste para a execução do teste de mutação para ontologias; (iii) definição de um conjunto de operadores de mutação que abordem os termos da linguagem OWL e sejam compatíveis com os tipos de defeitos ocasionados durante o desenvolvimento de ontologias; (iv) definição de um método padronizado para a geração dos dados de teste; (v) automatização da geração dos mutantes e dos dados de teste; e (vi) avaliação dos operadores de mutação.

1.5 ORGANIZAÇÃO DO TRABALHO

Esta tese está dividida em seis capítulos. No Capítulo 2 é apresentada a revisão bibliográfica com definições dos conceitos de ontologias, lógica descritiva, linguagem OWL e questões de competência. Ainda no Capítulo 2 é apresentada uma abordagem ao teste de mutação para programas, e na sequência os trabalhos relacionados e uma lista com os possíveis defeitos ocasionados no desenvolvimento de ontologias identificados na literatura. Finalizando este capítulo são apresentadas as considerações finais.

No Capítulo 3 são apresentadas as contribuições conceituais desta tese para o teste de mutação para ontologias OWL, sendo abordadas as definições de engano, defeito, erro e falha no contexto de ontologias. Na sequência são apresentadas as definições de dados e casos de teste para o teste de mutação para ontologias e os operadores de mutação propostos. Este capítulo é finalizado com as considerações finais.

No Capítulo 4 é apresentado o método de aplicação do teste de mutação para ontologias OWL, onde é abordada a ferramenta proposta para a geração dos mutantes e dos dados de teste, a execução do teste na ontologia original e nas ontologias mutantes e a análise dos resultados. Finalizando este capítulo são apresentadas as considerações finais.

No Capítulo 5 é abordada a avaliação do teste de mutação para ontologias OWL. São apresentados dois experimentos onde são descritos a motivação, método de execução, resultados e análise dos resultados para cada experimento. Este capítulo é finalizado com a análise geral dos dois experimentos e considerações do capítulo.

Finaizando esta tese, o Capítulo 6 apresenta as conclusões desta pesquisa, contribuições adicionais, limitações identificadas e trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA E CONCEITOS BÁSICOS

Neste capítulo é apresentada a revisão bibliográfica que fundamenta as contribuições desta tese. Na Seção 2.1 é apresentada a definição formal de ontologias, com o objetivo de compreender a sua composição e as possibilidades de mutações para auxiliar na definição dos operadores de mutação. Na sequência, na Subseção 2.1.1 é apresentada de forma resumida a lógica descritiva que é um formalismo de representação do conhecimento das ontologias e nesta tese é utilizada para a composição dos dados de teste. Adiante, na Subseção 2.1.2 é descrito como é composto um modelo OWL, com base na apresentação da Figura 2.5 do metamodelo de ontologias OWL. Na Subseção 2.1.3 é apresentada de forma sucinta a linguagem OWL para a construção de ontologias que sejam interpretadas por computadores. Finalizando a Seção 2.1, dada a variedade de representações que podem ser utilizadas para representar um único domínio em OWL, na Subseção 2.1.4 é apresentado o uso de Questões de Competência - CQ (do inglês, *Competency Question*) para auxiliar na avaliação de ontologias. Dada a definição de uma CQ, nesta tese elas são usadas na Seção 4.2 como uma formalização para a representação de um dado de teste, de modo que cada dado de teste é considerado uma questão de competência formalizada em lógica descritiva. Os trabalhos relacionados ao desenvolvimento desta tese são apresentados na Seção 2.3, onde são descritos os trabalhos referentes a avaliação e testes de ontologias na Subseção 2.3.1, e os tipos de defeitos ocasionados durante o desenvolvimento de ontologias na Subseção 2.3.2.

2.1 ONTOLOGIAS

Várias definições para o termo ontologia são encontradas na literatura, sendo a mais específica e explícita na área da Ciência da Computação, as definições apresentadas por Gruber (1995) e Borst (1997), que apontam que uma ontologia é uma especificação formal explícita de uma conceitualização compartilhada. O termo conceitualização refere-se a um modelo abstrato do domínio do conhecimento que se quer representar, de modo que essa representação deva ser explícita através da especificação de seus conceitos, propriedades e relacionamentos, para possibilitar o seu compartilhamento, permitindo assim que o conhecimento expresso seja do senso comum e não particular a quem está criando esse modelo (Prado, 2005).

Diante disso, as ontologias definem termos para descrever e representar domínios do conhecimento. Constituem-se em definições de conceitos, classes, propriedades, instâncias, relações, restrições e axiomas sobre um determinado domínio e padronizam significados através de identificadores semânticos para representar o mundo real e conceitual (Gruber, 1995; Freitas e Schulz, 2009). Elas requerem o uso de um vocabulário específico para descrever determinada realidade, sendo possível capturar os conceitos e relações em algum domínio, e um conjunto de axiomas permitem restringir a sua interpretação (Freitas e Schulz, 2009). Elas podem ser usadas como uma estrutura para representação semântica de informação (Davies et al., 2003; Gruber, 1995).

As classes de uma ontologia correspondem a um mecanismo de abstração para agrupar recursos com características similares, de modo que definem conjuntos de indivíduos que compartilham algumas propriedades. O conjunto de indivíduos que está associado a uma classe, é denominado como extensão de classe, de modo que os indivíduos em uma extensão de classe são chamados de instâncias da classe (Bechhofer et al., 2004; Gasevic et al., 2009). A estruturação das extensões de classe, bem como a da hierarquia na ontologia, ocorre através da aplicação

de um classificador (do inglês, *reasoner*), que é um programa que analisa expressões lógicas em um conjunto de axiomas explicitamente definidos, fornecendo suporte automatizado para classificação, depuração e consultas sobre a ontologia (Dentler et al., 2011).

Segundo Baader et al. (2003), Wimalasuriya e Dou (2009) e Korst et al. (2006), formalmente, uma ontologia O pode ser definida como uma quíntupla (C, P, I, V, A), onde:

(a) $C = (c_1, c_2, ..., c_n)$ é um conjunto de n **classes**, onde c_i é a i-ésima classe. As classes são um mecanismo de abstração para agrupar recursos com características similares, representando conjuntos de indivíduos que compartilham algumas propriedades (Gasevic et al., 2009).

(b) $P = (p_1(c_{1,1}, c_{1,2}), p_2(c_{2,1}, c_{2,2}), ..., p_m(c_{m,1}, c_{m,2}))$ é um conjunto de m **propriedades**, onde p_j é a j-ésima propriedade e $c_{j,1}$, $c_{j,2}$ são classes pertencentes a C, associadas por p_j.

(c) $I = (I_1, I_2, ..., I_m)$, onde I_i é o conjunto de **indivíduos ou instâncias da classe** c_i, denominado como extensão de classe (Gasevic et al., 2009).

(d) $V = (T_1, T_2, ..., T_m)$, onde T_j é um conjunto de **instâncias da propriedade** p_j, ou seja, pares (a, b) pertencentes ao produto cartesiano $I_{j,1}$ x $I_{j,2}$ de indivíduos das classes associadas pela propriedade p_j.

(e) A = Conjunto de axiomas definidos pela tripla (C, P, I):

(e.1) Tripla ou Axioma: Afirmação na forma *sujeito(s)-propriedade(p)-objeto(o)*. Um sujeito (C) é associado por uma propriedade (P) a um objeto (I). Para um conjunto (C, P, I), uma tripla (s, p, o) $\in$ ($C \sqcup P$) x P x ($C \sqcup P \sqcup I$) (Hayes e Patel-Schneider, 2014).

A Figura 2.1 apresenta um exemplo de uma tripla para descrever um objeto "*Motorista*". Nesta figura, o sujeito (s) "*Pessoa*" está associado através da propriedade (p) "*dirige*" a um objeto (o) "*Carro*". Deste modo, qualquer classe ou indivíduo pertencentes a "*Pessoa*" e que estejam associados através da propriedade "*dirige*" a outras classes ou indivíduos pertencentes a carro, serão classificados como "*Motorista*".

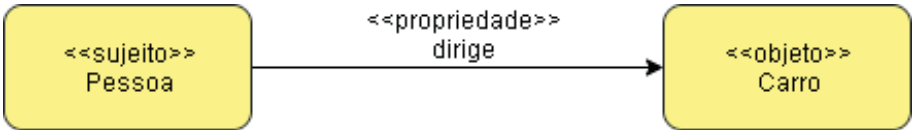


Figura 2.1: Exemplo de tripla sujeito-propriedade-objeto (autor, 2019)

As triplas dividem-se em dois grupos:

(e.1.1) Axiomas afirmativos (*ABox*): *C(a)*, *P(a,b)*, onde a e $b \in I$.

Na Figura 2.2 é apresentado um exemplo de um axioma *Abox* no qual dois indivíduos estão associados pela propriedade "*dirige*", representando de modo conceitual que o *IndivíduoA* pertencente a classe *Pessoa* dirige o *CarroB* que pertence a classe *Carro*, sendo possível aferir que o *IndivíduoA* é um motorista. Porém, através dessa afirmação não é possível aferir o inverso, dizer que o *CarroB* é dirigido ou conduzido pelo *IndivíduoA*.

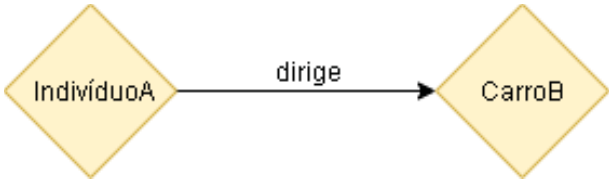


Figura 2.2: Exemplo de axioma Abox (autor, 2019)

(e.1.2) Axiomas terminológicos (*TBox*): $c_1 \sqsubseteq c_2$, onde c_1 é uma **subclasse** da classe c_2, ou $c_1 \equiv c_2$, onde a classe c_1 é **equivalente** a classe c_2.

Na Figura 2.3 A) é apresentado um exemplo onde todos os indivíduos pertencentes às classes *Homem* e *Mulher* são também pertencentes à classe *Pessoa*, dado que as classes *Homem* e *Mulher* são subclasses da classe *Pessoa*. Já na Figura 2.3 B), todos os indivíduos da classe *Pessoa* pertencem à classe *Humano* e todos os indivíduos da classe *Humano* também pertencem à classe *Pessoa*, dada a existência de uma definição de equivalência entre essas duas classes.

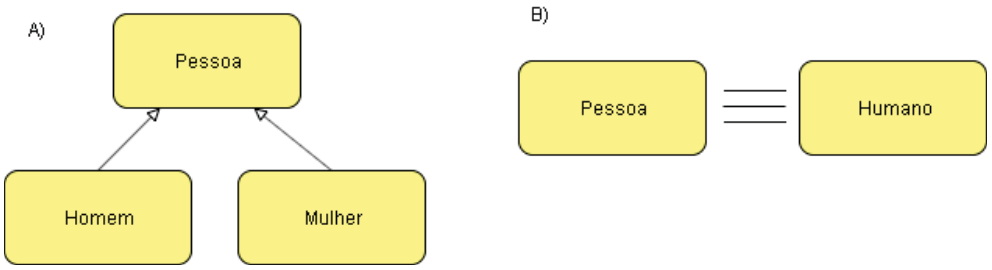


Figura 2.3: Exemplo de axioma Tbox (autor, 2019)

O potencial das ontologias encontra-se na habilidade de criar relacionamentos entre classes e instâncias e atribuir propriedades para esses relacionamentos, permitindo aplicar deduções sobre eles (Jepsen, 2009). Assim, é possível descrever relacionamentos com propriedades específicas entre as instâncias de cada classe, estabelecendo-se raciocínios sobre o domínio do conhecimento abordado. As classes de uma ontologia podem também ser subclasses de uma ou mais classes superiores chamadas superclasses. Desse modo, o raciocínio lógico descritivo definido em uma ontologia permite inferir que qualquer indivíduo de uma classe é considerado também como pertencente a todas as superclasses da qual sua classe deriva, conforme é apresentado na Figura 2.4.

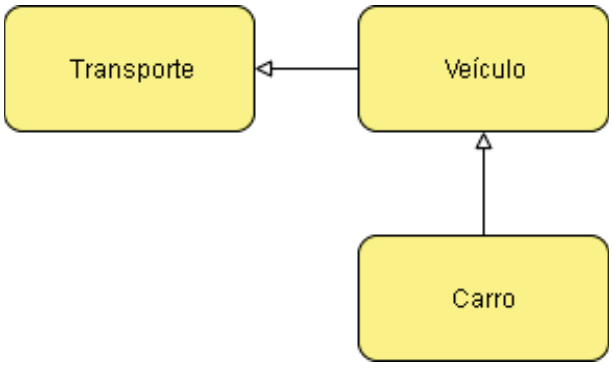


Figura 2.4: Instanciação de classes e superclasses (autor, 2019)

Conforme a Figura 2.4, é possível inferir que se todo *Veículo* é um tipo de *Transporte* e, se todo *Carro* é um tipo de *Veículo*, logo, todo *Carro* também é um tipo de *Transporte*.

Esta potencialidade de inferências e representação do conhecimento das ontologias é especificada em um formalismo da família de linguagens de Lógica Descritiva - DL (do inglês, *Description Logic*). As inferências em uma ontologia baseiam-se na suposição de mundo aberto, que conforme Reiter (1978), significa que uma ontologia não faz nenhuma suposição sobre a verdade ou falsidade de um fato a menos que possa ser provado. O formalismo mais básico para representação de uma ontologia é especificado pela linguagem DL $\mathcal{ALC}$ (do inglês, *Alternative Concept Language with Complements*), uma linguagem DL alternativa com um conjunto básico de características, como a definição de conceitos e indivíduos, negação, união, intersecção e restrições universal e existencial (Schmidt-Schauß e Smolka, 1991).

Na sequência, na Subseção 2.1.1 é apresentada a família de linguagens de lógica descritiva utilizadas para a representação do conhecimento em ontologias.

2.1.1 Lógica Descritiva

Conforme Baader et al. (2003), lógica descritiva é o nome dado para uma família de formalismos de representação do conhecimento de um domínio. A forma mais básica de representação de um domínio usando lógica descritiva, tem sido introduzida por Schmidt-Schauß e Smolka (1991) com a linguagem $\mathcal{ALC}$. Essa linguagem apresenta um conjunto minímo de elementos para representação do conhecimento, sendo eles a definição de conceitos atômicos, tautologia, contradição e intersecção representados pela lógica $\mathcal{AL}$ e a negação pela lógica $\mathcal{C}$. O Quadro 2.1 apresenta esses elementos e um exemplo de aplicação.

Quadro 2.1: Conjunto de elementos da lógica descritiva $\mathcal{ALC}$ adaptado de Schmidt-Schauß e Smolka (1991)

Elemento	**Descrição**	**Exemplo**
A	Conceito atômico	*Pessoa, Feminino* Descrição de dois conceitos atômicos distintos
$\top$	Tautologia	*Pessoa* $\sqcap \exists$temFilho.$\top$ | *Pessoa* $\sqcap \exists$hasChild.Female Descreve o conjunto de pessoas que têm no mínimo uma filha
$\bot$	Contradição	*Pessoa* $\sqcap \exists$temFilho.$\bot$ | *Pessoa* $\sqcap \exists$hasChild.Female Descreve o conjunto vazio de pessoas que têm filhos
¬A	Negação atômica	*Pessoa* $\sqcap \neg$*Feminino* Descreve as pessoas que não são do sexo feminino
C $\sqcap$ D	Intersecção	*Pessoa* $\sqcap$ *Feminino* Descreve as pessoas que são do sexo feminino

Para se obter uma linguagem mais expressiva, são adicionados mais elementos para a linguagem lógica $\mathcal{ALC}$. A união de conceitos é indicada pela lógica $\mathcal{U}$. A quantificação existencial é representada pela lógica $\mathcal{E}$ e restrições de cardinalidade são dadas pela lógica $\mathcal{N}$ (Baader e Nutt, 2003). Para a representação hierarquica de conceitos e propriedades é utilizada a lógica $\mathcal{H}$, para classes enumeradas a lógica $\mathcal{O}$ e com a lógica $\mathcal{I}$ é possível a definição de propriedades inversas (Baader et al., 2008; Hitzler et al., 2010). O Quadro 2.2 apresenta de forma resumida um comparativo entre a sintaxe da lógica descritiva e o seu tipo correspondente para cada elemento lógico.

Quadro 2.2: Representação da sintaxe *DL* (Baader et al., 2003, 2008)

Elemento	**Tipo *DL***	***DL***	**Exemplo**
Intersecção	$\mathcal{AL}$	$C \sqcap D$	*Pessoa* $\sqcap$ *Masculino*
União	$\mathcal{U}$	$C \sqcup D$	*Pessoa* $\sqcup$ *Masculino*
Negação	$\mathcal{C}$	$\neg C$	$\neg$*Masculino*
Classes enumeradas	$\mathcal{O}$	{a} ... {b}	{joão, maria}
Restrição existencial	$\mathcal{E}$	$\exists\, r.C$	$\exists$dirige.Carro
Restrição universal	$\mathcal{AL}$	$\forall\, r.C$	$\forall$dirige.Veículo
Cardinalidade mínima	$\mathcal{N}$	$\geq NR$	($\geqslant$ 2 temFilho)
Cardinalidade máxima	$\mathcal{N}$	$\leq NR$	($\leqslant$ 1 temFilho)
Igualdade	$\mathcal{N}$	$= NR$	(= 1 temFilho)
Restrição de valor	$\mathcal{O}$	$\exists R.\{a\}$	$\exists$nacionalDo.Brasil
Subclasse	$\mathcal{H}$	$C \sqsubseteq D$	*Homem* $\sqsubseteq$ *Pessoa*
Equivalência	$\mathcal{H}$	$C \equiv D$	*Homem* $\equiv$ *Pessoa* $\sqcap$ *Masculino*
Disjunção	$\mathcal{C}$	$C \sqsubseteq \neg D$	*Homem* $\sqsubseteq$ $\neg$*Mulher*
Equivalência entre indivíduos	$\mathcal{H}$	$\{a\} \equiv \{b\}$	{asteroide} $\equiv$ {meteoro}
Disjunção entre indivíduos	$\mathcal{C}$	$\{a\} \equiv \neg\{b\}$	{joão} $\sqsubseteq$ $\neg${maria}

Como a lógica descritiva provê um formalismo lógico para linguagens de ontologias, a linguagem OWL definida pelo W3C *Web-Ontology Working Group*[1] como uma linguagem declarativa para construção de ontologias de forma lógica, tem como uma de suas características ser formalizada em lógica descritiva. Conforme Baader et al. (2008), a linguagem OWL é definida através do mapeamento da lógica descritiva. Esse mapeamento permite que a linguagem OWL explore os resultados da pesquisa em lógica descritiva com relação à decidibilidade e complexidade dos principais problemas de inferência.

2.1.2 Modelos OWL

Com o crescimento da *Web* Semântica e a possibilidade de uso das ontologias na interoperabilidade de sistemas, o consórcio *World Wide Web* (W3C)[2] criou a linguagem OWL como a linguagem padrão para a construção de ontologias. A OWL é uma linguagem de marcação semântica para publicação e compartilhamento de ontologias, projetada para descrever classes e relacionamentos entre elas (McGuinness e van Harmelen, 2004). A Linguagem OWL foi projetada para representar um conhecimento rico e complexo sobre os conceitos do domínio, grupos de conceitos e as relações entre eles. É uma linguagem formalizada em lógica descritiva computacional, de modo que o conhecimento expresso em OWL possa ser explorado por sistemas computacionais, tais como verificar a consistência do conhecimento ou tornar explícito um conhecimento implícito (Baader et al., 2003).

A linguagem OWL adota um formalismo da orientação a objetos no qual os conceitos do domínio são descritos em termos de classes, propriedades e indivíduos (Horrocks, 2008), de acordo com a definição de ontologias apresentada na Seção 2.1. Conforme Gasevic et al. (2009), esse modelo é abstraído do Metamodelo de Definição de Ontologias - ODM (do inglês, *Ontology Definition Metamodel*). Esse metamodelo[3] apresenta dois modelos para representar ontologias. O primeiro modelo é definido pela linguagem RDFS (do inglês, *Resource Description*

[1] https://www.w3.org/TR/owl-ref/

[2] https://www.w3.org/

[3] Modelo explícito de construtores e regras necessárias para construir modelos específicos dentro de um domínio de interesse.

Framework Schema) que é uma linguagem de descrição de vocabulários que objetiva descrever propriedades e classes para os recursos RDF, que por sua vez é uma linguagem para representar objetos na *web*. O metamodelo RDFS é uma abstração do modelo RDF, que além de representar seus conceitos, define também o modelo OWL. A descrição de um objeto em RDF é uma tripla na forma sujeito-propriedade-objeto (Gasevic et al., 2009).

O segundo modelo do metamodelo OWL é definido pela própria linguagem OWL, que descreve relacionamentos semânticos entre objetos. Desse modo, OWL possibilita descrever relações lógicas descritivas entre classes, propriedades e indivíduos (Gasevic et al., 2009). Como OWL estende os conceitos do metamodelo RDFS, também é definida através de triplas, conforme o modelo RDF. O metamodelo OWL é apresentado na Figura 2.5. De acordo com este metamodelo e a definição formal de ontologias apresentada na Seção 2.1, as ontologias OWL são compostas por classes (*OWLClass*), propriedades (*Property*) e indivíduos (*Individual*).

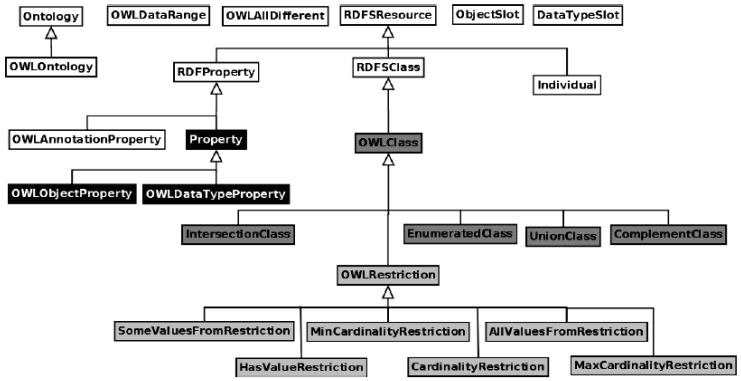


Figura 2.5: Metamodelo de ontologias OWL (Gasevic et al., 2009)

A Figura 2.6 A) apresenta a estrutura de classes da ontologia *travel*[4] (Knublauch, 2003), uma ontologia que descreve o domínio do turismo. Na Figura 2.6 B) são apresentadas as propriedades de objetos e de tipos de dados dessa ontologia e na Figura 2.6 C) os invidívuos.

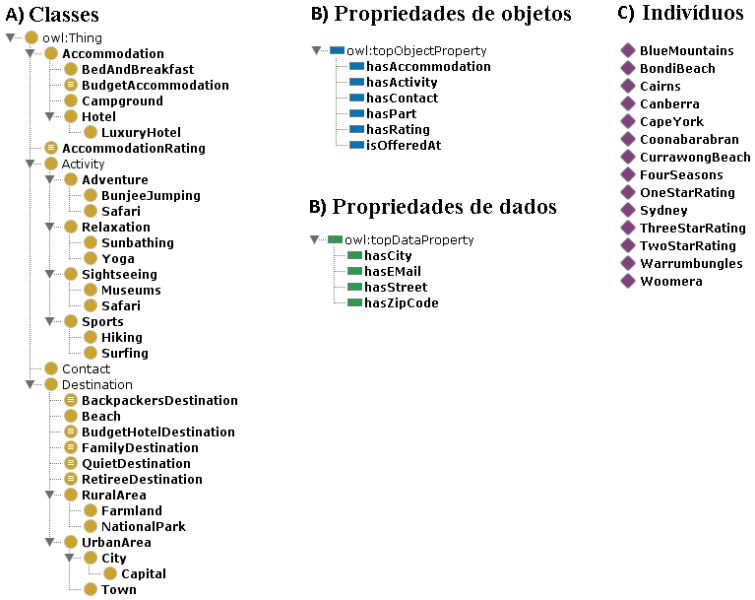


Figura 2.6: Classes, propriedades e indivíduos da ontologia *travel* (Knublauch, 2003)

[4]https://protege.stanford.edu/ontologies/travel.owl

As classes podem ser construídas como complemento, união ou intersecção de outras classes utilizando a notação de lógica descritiva conforme apresentada no Quadro 2.2, de modo que possibilitam a definição de um conjunto de restrições de acordo com a lógica do domínio abordado. O Exemplo 2.1 apresenta a notação em lógica descritiva da definição de equivalência da classe *QuietDestination* da ontologia *travel* utilizando intersecção e união.

Exemplo 2.1: Definição de equivalência da classe *QuietDestination* em notação de lógica descritiva (autor, 2019)

```
Destination ⊓ (¬(FamilyDestination))
```

De acordo com o Exemplo 2.1, a extensão da classe *QuietDestination* é a mesma extensão da classe representada pelos indivíduos que compõem a classe resultante da disjunção entre as classes *Destination* e *FamilyDestination*.

A hierarquia de classes é representada através da definição de que uma classe é subclasse de outra, indicando que uma subclasse é um tipo da sua superclasse, ou seja, todas as instâncias de uma classe pertencem também a sua superclasse. Em um conjunto de duas classes, sendo as classes *LuxuryHotel* e *Hotel* da ontologia *travel*, é possível representar que a classe *LuxuryHotel* é subclasse da classe *Hotel* utilizando a notação de lógica descritiva conforme o Exemplo 2.2.

Exemplo 2.2: Representação de subclasse em notação de lógica descritiva (autor, 2019)

```
LuxuryHotel ⊑ Hotel
```

As propriedades da ontologia podem ser tanto propriedades de tipos de dados, responsáveis por associar um indivíduo a um tipo de dado literal, ou propriedades de objetos, responsáveis por associar um indivíduo a outro. Esse modo de representação das ontologias OWL possibilita que o desenvolvedor especifique inferências lógicas descritivas na ontologia, de modo a derivar consequências lógicas através da semântica formal OWL, esclarecendo fatos que não estão presentes na ontologia, mas são vinculados pela semântica (Smith et al., 2004).

Deste modo, uma ontologia OWL consiste de um conjunto de expressões lógicas que representam axiomas definidos em lógica descritiva para representar um domínio específico do conhecimento (Horrocks, 2008). Conforme Bechhofer et al. (2004), os axiomas descrevem as relações entre os elementos de uma ontologia OWL, sejam eles classes, propriedades ou indivíduos. Conforme Rodes e Pospesel (1996) e Hitzler et al. (2010), uma expressão lógica consiste de no mínimo uma premissa e uma conclusão, com o significado intuitivo de que, em qualquer situação em que a premissa se aplica, a conclusão deve também ser válida. Tal descrição geral compreende os axiomas OWL. Desse modo, uma ontologia OWL pode ser considerada como um conjunto de axiomas que através dos construtores OWL permitem a instanciação de objetos em classes e subclasses, dos quais se aplicam em toda a estrutura das ontologias. Nesse contexto, a classe *BudgetAccommodation* da ontologia *travel*, é descrita através do axioma lógico descritivo apresentado no Exemplo 2.3.

Exemplo 2.3: Axioma lógico descritivo da classe *BudgetAccommodation* da ontologia *travel* (autor, 2019)

```
Accommodation ⊓ (∃hasRating.(oneStarRating, twoStarRating))
```

O conceito "*Accommodation*" do Exemplo 2.3 representa uma classe, o conceito "*hasRating*" representa uma propriedade de objeto responsável por realizar a associação entre a classe "*Accommodation*" e todos os demais objetos associados no mínimo uma vez com os indivíduos *oneStarRating*" e "*twoStarRating*". É possível observar que todas as instâncias da classe *BudgetAccommodation* referem-se a todos os indivíduos que pertencem à classe *Accommodation* e estão associados ao mesmo tempo aos indivíduos *oneStarRating* ou *twoStarRating*.

A linguagem OWL possibilita que relacionamentos mais complexos entre as classes e propriedades sejam descritos. Esses relacionamentos são modelados por meio de uma série de

construtores da linguagem OWL que são mapeados da lógica descritiva. Na Subseção 2.1.3 são aprsentados esses construtores e a sintaxe da linguagem OWL para construção de ontologias.

2.1.3 Linguagem OWL

Uma das principais características da linguagem OWL é ser formalizada em uma DL expressiva denominada $\mathcal{SHOIN}(\mathcal{D})$, que aborda tanto *Tbox* quanto *Abox*. Essa DL é uma abreviação de um acrônimo derivado de várias características da lógica descritiva, onde o $\mathcal{S}$ é uma abreviação para $\mathcal{ALC}$ e $(\mathcal{D})$ representa tipos de dados (Hitzler et al., 2010). A linguagem OWL apresenta construtores específicos para a definição de classes, propriedades e indivíduos. Para a construção de axiomas lógico-descritivos, responsáveis pela definição lógica e conceitual da ontologia, OWL realiza um mapeamento dos elementos de lógica descritiva em uma série de construtores OWL. O Quadro 2.3 apresenta o comparativo entre os elementos da lógica descritiva e dos respectivos construtores OWL.

Quadro 2.3: Representação da sintaxe *DL* em *OWL* (Horridge et al., 2006; Baader et al., 2003)

Elemento	***DL***	**OWL**
Intersecção	$C \sqcap D$	*intersectionOf*
União	$C \sqcup D$	*unionOf*
Negação	$\neg C$	*complementOf*
Classes enumeradas	$\{a\} \sqcup \{b\}$	*oneOf*
Restrição existencial	$\exists\, r.C$	*someValuesFrom*
Restrição universal	$\forall\, r.C$	*allValuesFrom*
Cardinalidade mínima	$\geq NR$	*minCardinality*
Cardinalidade máxima	$\leq NR$	*maxCardinality*
Igualdade	$= NR$	*cardinality*
Restrição de valor	$\exists\, R\ \{a\}$	*hasValue*
Subclasse	$C \sqsubseteq D$	*subClassOf*
Equivalência	$C \equiv D$	*equivalentOf*
Disjunção	$C \sqsubseteq \neg D$	*disjointWith*
Equivalência entre indivíduos	$\{a\} \equiv \{b\}$	*sameAs*
Disjunção entre indivíduos	$\{a\} \equiv \neg\{b\}$	*differentFrom*

A sintaxe da linguagem OWL é especificada como uma serialização da linguagem XML (do inglês, *eXtensible Markup Language*). Os blocos básicos de construção são as classes, que são definidas em OWL utilizando a notação *owl:Class*, as propriedades, que podem ser tanto propriedades de objetos, responsáveis por associar um objeto a outro, denotadas pela notação *owl:ObjectProperty* e as propriedades de dados, responsáveis por associar um objeto a um tipo de dado literal, sendo representadas pela notação *owl:DataTypeProperty*. Para a definição de indivíduos, OWL apresenta a notação *owl:NamedIndividual*. O Exemplo 2.4 apresenta a sintaxe de representação da classe *Accommodation*, da propriedade de objetos *hasAccommodation*, da propriedade de dados *hasCity* e do indivíduo *Sydney* da ontologia *travel* da Figura 2.6[5].

Exemplo 2.4: Definição de classes, propriedades e indivíduos em OWL (autor, 2019)

```
<owl:Class rdf:about=''Accommodation'' />
<owl:ObjectProperty rdf:about=''hasAccommodation'' />
<owl:DataTypeProperty rdf:about=''hasCity'' />
<owl:NamedIndividual rdf:about=''Sydney''>
```

[5]Os próximos exemplos até o Exemplo 2.19 referem-se a ontologia *travel* da Figura 2.6

2.1.3.1 Relações de classes

As classes OWL são descritas como subclasses, equivalentes ou como disjuntas a outras classes. Uma classe é definida como subclasse de outra através da notação *rdfs:subClassOf*. O Exemplo 2.5 apresenta a sintaxe da classe *Hotel* como subclasse da classe *Accommodation*.

Exemplo 2.5: Definição de subclasse em OWL (autor, 2019)

```
<owl:Class rdf:about=''Hotel''>
    <rdfs:subClassOf rdf:resource=''Accommodation'' />
</owl:Class>
```

Para uma relação de classe e subclasse mais complexa, OWL especifica uma classe anônima que representa o conjunto de instâncias que fazem parte de uma relação. Essa relação é descrita através de um axioma lógico descritivo. O Exemplo 2.6 apresenta a classe *LuxuryHotel* descrita como subclasse do conjunto de indivíduos associados pela propriedade de objetos *hasRating* ao indivíduo *ThreeStarRating*. Essa relação indica que um hotel é considerado como hotel de luxo quando possuir três estrelas.

Exemplo 2.6: Definição de subclasse de um conjunto de indivíduos em OWL (autor, 2019)

```
<owl:Class rdf:about=''LuxuryHotel''>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''hasRating'' />
            <owl:hasValue rdf:resource=''ThreeStarRating'' />
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Para a definição de equivalência entre classes é utilizada a notação *owl:equivalentClass*. Assim como ocorre para a representação de subclasses, a equivalência pode ser definida entre duas ou mais classes ou através da representação de um axioma lógico descritivo. O Exemplo 2.7 apresenta a sintaxe de representação da classe *AccommodationRating* como equivalente ao conjunto de indivíduos *OneStarRating* e *TwoStarRating*.

Exemplo 2.7: Definição de equivalência em OWL (autor, 2019)

```
<owl:Class rdf:about=''AccommodationRating''>
    <owl:equivalentClass>
        <owl:Class>
            <owl:oneOf rdf:parseType=''Collection''>
                <rdf:Description rdf:about=''OneStarRating'' />
                <rdf:Description rdf:about=''TwoStarRating'' />
            </owl:oneOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>
```

Para a especificação de que duas ou mais classes são disjuntas é utilizada a notação *owl:disjointWith*. O Exemplo 2.8 apresenta a sintaxe de representação de disjunção entre as classes *UrbanArea* e *RuralArea*, de modo que um objeto instanciado como um tipo de área urbana não possa ser representado como um tipo de área rural ao mesmo tempo.

Exemplo 2.8: Definição de disjunção em OWL (autor, 2019)

```
<owl:Class rdf:about=''UrbanArea''>
    <rdfs:subClassOf rdf:resource=''Destination'' />
    <owl:disjointWith rdf:resource=''RuralArea'' />
</owl:Class>
```

2.1.3.2 Relações de indivíduos

A linguagem OWL também permite declarar que dois indivíduos nomeados de forma diferente são de fato o mesmo indivíduo utilizando a notação *owl:sameAs*, similar à equivalência entre classes. O Exemplo 2.9 apresenta os indivíduos *Sydney* e *NovaGalesCapital* como sendo o mesmo indivíduo, dado que *Sydney* é a capital do estado de Nova Gales do Sul.

Exemplo 2.9: Definição de equivalência entre indivíduos em OWL (autor, 2019)

```
<rdf:Description rdf:about=''Sydney''>
    <owl:sameAs rdf:resource=''NovaGalesCapital'' />
</rdf:Description>
```

Similar à disjunção entre classes, OWL infere que um indivíduo é diferente de outro através da notação *owl:distinctMembers*. O Exemplo 2.10 apresenta a sintaxe para representar que os indivíduos *OneStarRating* e *TwoStarRating* são diferentes entre si.

Exemplo 2.10: Definição de disjunção entre indivíduos em OWL (autor, 2019)

```
<rdf:Description>
    <rdf:type rdf:resource=''AllDifferent'' />
    <owl:distinctMembers rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''OneStarRating'' />
        <rdf:Description rdf:about=''TwoStarRating'' />
    </owl:distinctMembers>
</rdf:Description>
```

2.1.3.3 Relações booleanas

Para expressar uma representação mais complexa, a linguagem OWL fornece construtores lógicos para classes. Esses construtores são expressos pela notação *owl:intersectionOf*, *owl:unionOf* e *owl:complementOf*, representando respectivamente os elementos lógicos *and*, *or* e *not* para expressar a conjunção, disjunção e negação. A conjunção entre duas classes consiste de exatamente aqueles objetos que pertencem a ambas as classes. O Exemplo 2.11 apresenta a classe *QuietAccommodation* como equivalente às classes *Destination* e *FamilyDestination*.

Exemplo 2.11: Definição de conjunção em OWL (autor, 2019)

```
<owl:Class rdf:about=''QuietDestination''>
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType=''Collection''>
                <rdf:Description rdf:about=''Destination'' />
                <rdf:Description rdf:about=''FamilyDestination'' />
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>
```

A disjunção lógica entre duas classes, representada pela notação *owl:unionOf*, expressa que um objeto pertence a no mínimo uma das duas classes. O Exemplo 2.12 apresenta a definição conceitual da classe *BackPackersDestination* como equivalente às classes *Adventure* ou *Sports*.

Exemplo 2.12: Definição de disjunção lógica em OWL (autor, 2019)

```
<owl:Class rdf:about=''BackPackersDestination''>
    <owl:equivalentClass>
        <owl:Class>
            <owl:unionOf rdf:parseType=''Collection''>
                <rdf:Description rdf:about=''Adventure'' />
                <rdf:Description rdf:about=''FamilyDestination'' />
            </owl:unionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>
```

O complemento de uma classe corresponde à negação lógica e consiste em especificar os objetos que não são membros dessa classe. Com base no Exemplo 2.11, poderia ser especificado que a classe *QuietAccommodation* é equivalente à classe *Destination*, porém não aos indivíduos membros da classe *FamilyDestination*. O Exemplo 2.13 apresenta a sintaxe dessa representação.

Exemplo 2.13: Definição de negação em OWL (autor, 2019)

```
<owl:Class rdf:about=''QuietDestination''>
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType=''Collection''>
                <rdf:Description rdf:about=''Destination'' />
                <owl:Class>
                    <owl:complementOf rdf:resource=''FamilyDestination'' />
                </owl:Class>
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>
```

2.1.3.4 Relações de propriedades

Ao definir uma propriedade para associação de objetos, a linguagem OWL possibilita especificar o domínio e intervalo dessa propriedade através das notações *owl:Domain* e *owl:Range*. O domínio (*owl:Domain*) permite inferir uma classificação de assuntos que essa propriedade se refere. Já, ao especificar o intervalo (*owl:Range*), é possivel determinar quais objetos coincidem com essa propriedade. As restrições *owl:Domain* e *owl:Range* representam um link semântico entre classes e propriedades, porque essas restrições são o único modo de descrever as relações terminológicas entre os diferentes tipos de elementos da ontologia. O Exemplo 2.14 apresenta a sintaxe para definição do domínio e intervalo da propriedade de objetos *hasAccommodation*.

Exemplo 2.14: Definição de domínio e *range* de propriedades em OWL (autor, 2019)

```
<owl:ObjectProperty rdf:about=''hasAccommodation''>
    <rdfs:domain rdf:resource=''Destination'' />
    <rdfs:range rdf:resource=''Accommodation'' />
</owl:ObjectProperty>
```

Conforme o Exemplo 2.14, é possível inferir que as associações realizadas pela propriedade *hasAccommodation* referem-se aos tipos de acomodações existentes nos diferentes

tipos de destinos representados na ontologia, dado que o intervalo dessa propriedade é a classe *Accommodation* e o domínio é a classe *Destination*.

As propriedades permitem a construção de outros tipos de classes complexas, através do uso de elementos como por exemplo os quantificadores universal, existencial e restrições de cardinalidades. O quantificador universal, denotado por *owl:allValuesFrom* define um axioma lógico que representa uma classe como o conjunto de todos os objetos que sejam satisfeitos pelo intervalo definido na relação da propriedade. O Exemplo 2.15 apresenta a sintaxe para representar que a classe *RetireeDestination* é equivalente as acomodações do tipo três estrelas.

Exemplo 2.15: Definição de axioma universal em OWL (autor, 2019)

```
<owl:Class rdf:about=''RetireeDestination''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''hasAccommodation'' />
            <owl:allValuesFrom>
                <owl:Restriction>
                    <owl:onProperty rdf:resource=''hasRating'' />
                    <owl:hasValue rdf:resource=''ThreeStarRating'' />
                </owl:Restriction>
            </owl:allValuesFrom>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Para representar na ontologia que qualquer "destino" para "aposentados" deve ter no mínimo uma acomodação do tipo três estrelas, é possível representar o mesmo axioma do Exemplo 2.15 utilizando a notação existencial *owl:someValuesFrom*. O Exemplo 2.16 apresenta a sintaxe para representar que a classe *RetireeDestination* é equivalente as acomodações que possuem no mínimo três estrelas.

Exemplo 2.16: Definição de axioma existencial em OWL (autor, 2019)

```
<owl:Class rdf:about=''RetireeDestination''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''hasAccommodation'' />
            <owl:someValuesFrom>
                <owl:Restriction>
                    <owl:onProperty rdf:resource=''hasRating'' />
                    <owl:hasValue rdf:resource=''ThreeStarRating'' />
                </owl:Restriction>
            </owl:someValuesFrom>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Também é possível descrever axiomas com restrições de cardinalidade na linguagem OWL. Com a notação *owl:cardinality* pode-se dizer que a classe *FamilyDestination* que representa o conjunto de todos os objetos que são considerados destinos de turismo para famílias, deve possuir exatamente duas atividades diferentes. O Exemplo 2.17 apresenta a definição do axioma que descreve essa classe.

Exemplo 2.17: Definição de cardinalidade em OWL (autor, 2019)

```
<owl:Class rdf:about=''FamilyDestination''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''hasActivity'' />
            <owl:qualifiedCardinality
                rdf:datatype=''nonNegativeInteger''>2
            </owl:qualifiedCardinality>
            <owl:onClass rdf:resource=''Activity'' />
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

É possível representar o Exemplo 2.17 informando que para um objeto ser considerado um "destino de família", esse objeto deve estar associado a no mínimo 2 atividades diferentes. O Exemplo 2.18 apresenta a descrição da classe *FamilyDestination* como equivalente a objetos que estejam associados a no mínimo 2 atividades diferentes.

Exemplo 2.18: Definição de cardinalidade mínima em OWL (autor, 2019)

```
<owl:Class rdf:about=''FamilyDestination''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''hasActivity'' />
            <owl:minQualifiedCardinality
                rdf:datatype=''nonNegativeInteger''>2
            </owl:minQualifiedCardinality>
            <owl:onClass rdf:resource=''Activity'' />
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Para representar o Exemplo 2.18 de modo que a classe *FamilyDestination* seja composta por objetos que estejam associados a no máximo duas atividades diferentes, basta substituir o construtor *owl:minQualifiedCardinality* pelo construtor *owl:maxQualifiedCardinality*.

Com base nos diferentes tipos de associações e representações de conceitos que podem ser descritos em OWL, um domínio do conhecimento pode ser representado por diferentes ontologias, que podem resultar em diferentes estruturas, axiomas e consequentemente, diferentes resultados quando classificando ou consultando informação. Como um método para identificar se os axiomas da ontologia foram representados de forma satisfatória, Obrst et al. (2007) propôs a execução de consultas na ontologia definidas como questões de competência, com o objetivo de analisar o resultado esperado com o resultado obtido. Na próxima subseção é apresentado o conceito e como são formalizadas as questões de competência.

2.1.4 Questões de competência

Conforme Grüninger e Fox (1995), Questões de Competência - CQ (do inglês, *Competency Questions*), são um conjunto de requisitos que descrevem os tipos de conhecimentos que a ontologia retorna ao responder tais questões. As CQs tendem a direcionar o desenvolvedor durante o desenvolvimento da ontologia. Elas correspondem a perguntas formuladas em linguagem natural e expressam um padrão para um tipo de perguntas que a ontologia seja capaz de responder. A capacidade de resposta das CQs torna-se um requisito funcional da ontologia (Uschold e Gruninger, 1996). Uma ontologia permite a inclusão de novas CQs em qualquer momento do seu ciclo de vida.

Para que as CQs possam ser utilizadas com o objetivo de avaliação das ontologias, elas devem ser formalizadas em uma linguagem de consulta, para que os sistemas testem automaticamente se a ontologia relaciona os resultados com as questões de competência (Vrandečić e Gangemi, 2006). Para Bezerra et al. (2014), as CQs são formadas por um par composto pela questão e sua resposta, conforme segue a definição:

- **Questão de competência:** uma CQ é um par (*Q, q*), onde *Q* corresponde a uma consulta escrita em uma linguagem formal e, *q* é uma resposta para essa consulta.
- **Questão de competência satisfatória:** uma CQ satisfaz uma ontologia *O* se $O \models q(Q)$, ou seja, se *q(Q)* satisfaz *O*.

Conforme Vrandečić e Gangemi (2006), as CQs geralmente são questões representativas, não significando que ao responder todas as questões a ontologia esteja completa e, que não necessariamente as respostas devam ser conhecidas no momento da elaboração das questões. Algumas vezes as respostas das CQs podem ser simplesmente *Verdadeiro* ou *Falso*, em outros casos podem ser uma classe, lista de classes, uma instância ou uma lista de instâncias (García-Ramos et al., 2016). Para respostas booleanas como *verdadeiro* ou *falso*, uma CQ pode ser formulada conforme apresentada no exemplo 2.19:

Exemplo 2.19: Questão de competência com resposta booleana (autor, 2019)

```
Tontura é um sintoma de infarto?
```

Para questões cujas respostas consistem em classes e instâncias, uma CQ pode ser formulada conforme o exemplo 2.20:

Exemplo 2.20: Questão de competência com resposta consistindo em classes e instâncias (autor, 2019)

```
Quais os tipos de reações alérgicas são constatadas em gestantes?
```

As CQs devem ser traduzidas para linguagens de consultas para ontologias para que possam ser processadas e o resultado ser retornado pela ontologia. Considerando-se obter CQs de fácil interpretação e que possam ser executadas automaticamente através de uma ferramenta computacional como a ferramenta *Protégé* (Musen, 2015), as CQs podem ser formalizadas utilizando a sintaxe *OWL Manchester* (Horridge et al., 2006), que é uma sintaxe de fácil interpretação para a linguagem OWL e que se baseia fundamentalmente na coleta de dados e informações sobre uma determinada classe, propriedade ou indivíduo em uma única construção. O Quadro 2.4 apresenta a sintaxe *OWL Manchester* de cada componente da sintaxe DL para a descrição de classes em OWL.

Quadro 2.4: Representação da sintaxe *DL* em *OWL Manchester* (Horridge et al., 2006; Baader et al., 2003)

Construtor OWL	***DL***	***OWL Manchester***	**Exemplo**	**Tipo *DL***
intersectionOf	$C \sqcap D$	*C AND D*	*Pessoa* ***and*** *Feminino*	$\mathcal{AL}$
unionOf	$C \sqcup D$	*C OR D*	*Feminino* ***or*** *Masculino*	$\mathcal{U}$
complementOf	$\neg C$	*NOT C*	***not*** *Masculino*	$\mathcal{C}$
oneOf	{a} ⊔ {b}	{a b ...}	{*Brasil Chile Argentina*}	$\mathcal{O}$
someValuesFrom	$\exists$ *R C*	*R SOME C*	*autorDe* ***some*** *Livro*	$\mathcal{E}$
allValuesFrom	$\forall$ *R C*	*R ONLY C*	*autorDe* ***only*** *Livro*	$\mathcal{AL}$
minCardinality	$\geq$ *N R*	*R MIN 3*	*autorDe* ***min*** *3*	$\mathcal{N}$
maxCardinality	$\leq$ *N R*	*R MAX 3*	*autorDe* ***max*** *3*	$\mathcal{N}$
cardinality	$=$ *N R*	*R EXACTLY 3*	*autorDe* ***exactly*** *3*	$\mathcal{N}$
hasValue	$\exists$ *R* {a}	*R VALUE* {a}	*autorDe* ***value*** *Senhora*	$\mathcal{O}$

Considerando representar as CQs dos exemplos 2.19 e 2.20 em lógica descritiva utilizando a sintaxe *OWL Manchester* para serem executadas na ferramenta *Protégé*, de acordo com o Quadro 2.4 a CQ do Exemplo 2.19 seria descrita conforme o Exemplo 2.21.

Exemplo 2.21: CQ do Exemplo 2.19 representada em DL na sintaxe *OWL Manchester* (autor, 2019)

```
temSintoma some Tontura
```

Conforme o Exemplo 2.21, *temSintoma* refere-se a uma propriedade de objetos e *Tontura* a uma classe, de modo que ao executar essa CQ na ontologia, é possível identificar se existem indivíduos instanciados para a classe *Infarto* que estejam associados ao sintoma *Tontura*.

Já a CQ do Exemplo 2.20 é descrita conforme o Exemplo 2.22.

Exemplo 2.22: CQ do Exemplo 2.20 representada em DL na sintaxe *OWL Manchester* (autor, 2019)

```
Gestantes and temReações some ReaçõesAlérgicas
```

Conforme o Exemplo 2.22, *Gestantes* e *ReaçõesAlérgicas* são reprensentações de classes e, *temReações* é uma propriedade de objetos que associa os tipos de reações alérgicas às gestantes. Ao executar essa CQ na ontologia, é possível identificar todos os tipos de reações alérgicas associadas às gestantes.

2.2 TESTE DE MUTAÇÃO PARA PROGRAMAS

O teste de mutação é um critério de teste da técnica baseada em defeitos, onde são utilizados defeitos típicos do processo de implementação de software para que sejam derivados os requisitos de teste (Delamaro et al., 2007). Nessa técnica de teste, o código fonte de um programa *P* a ser testado é analisado (Lee et al., 2008), de modo que esse programa é alterado diversas vezes por operadores de mutação específicos, como se defeitos estivessem sendo inseridos no programa original, criando-se um conjunto de programas alternativos chamado mutantes (Delamaro et al., 2007). O Exemplo 2.23 apresenta a geração de um mutante pela aplicação de um operador de mutação que substitui o operando "<=" pelo operando ">=" no código fonte de um programa.

Exemplo 2.23: Geração de mutantes

```
CÓDIGO FONTE ORIGINAL
if (x <= y) {
    z = 10;
} else {
    z = 5;
}
```

```
CÓDIGO FONTE MUTANTE
  if (x >= y) {
    z = 10;
} else {
    z = 5;
}
```

O trabalho do testador é escolher casos de teste que mostrem a diferença de comportamento entre o programa original e o programa mutante (Delamaro et al., 2007), analisando-se a saída dos dois programas com o mesmo caso de teste.

Conforme DeMillo et al. (1978), o teste de mutação é baseado em dois pressupostos. O primeiro pressuposto é a hipótese do programador competente, na qual programadores experientes escrevem programas corretos ou muito próximos do correto. Assumindo a validade dessa hipótese, pode-se afirmar que erros são introduzidos nos programas através de pequenos desvios sintáticos que podem alterar a semântica do programa e, consequentemente, conduzir o programa a um comportamento incorreto (Agrawal et al., 1989). O segundo pressuposto é o efeito de acoplamento, onde assume-se que erros complexos estão relacionados a erros simples (Agrawal et al., 1989), de modo que conjuntos de casos de teste capazes de revelar erros simples são também capazes de revelar erros complexos (Maldonado et al., 1998).

Um caso de teste corresponde a um par formado por um dado de teste *T* mais o resultado esperado para a execução de *P* com aquele dado de teste. O dado de teste é um elemento do domínio de entrada de *P* (Delamaro et al., 2007), onde para um programa (Exemplo 2.23) que computa "x" e "y", podem ser considerados os seguintes casos de teste ((2,4),10), ((3,8),10), ((5,-2),5). Deste modo, *P* é executado com *T* e, se *P* apresentar resultados inesperados, então um defeito foi encontrado. Caso contrário, o programa ainda pode conter erros que o conjunto *T* não conseguiu revelar. O programa *P* sofre então pequenas alterações sintáticas, dando origem a um conjunto de programas mutantes, chamado *P'*, diferindo de *P* apenas pela ocorrência de erros simples (Agrawal et al., 1989). Essas pequenas alterações sintáticas são introduzidas em *P* através de operadores de mutação (DeMillo et al., 1978). Para executar cada programa pertencente a *P'*, é utilizado o mesmo conjunto *T* aplicado em *P*.

Tradicionalmente, não existe uma maneira direta de definir os operadores de mutação para uma dada linguagem. Em geral, os operadores de mutação são projetados tendo por base a experiência no uso de determinada linguagem, bem como os enganos, tipo ou classes de erros mais comuns cometidos durante a sua utilização e que se pretende revelar em *P* (Delamaro et al., 2007; Fabbri et al., 1999). Além dos tipos de defeitos que se deseja revelar e da cobertura que se quer garantir, a escolha de um conjunto de operadores de mutação depende também da linguagem em que os programas a serem testados estão escritos.

Após a execução de *P*, cada um dos programas mutantes *p* pertencentes ao conjunto de programas mutantes *P'* é executado com o mesmo conjunto de casos de teste *T* utilizados em *P*. Se um mutante *p* apresenta resultado diferente de *P*, isso demonstra que os casos de teste utilizados são eficientes e conseguiram revelar a diferença entre *P* e *p*, determinando que o programa mutante *p* está morto e é descartado (Delamaro et al., 2007). O teste de mutação exige que todos os mutantes sejam mortos (DeMillo et al., 1978). Porém, é possível que *p* apresente saídas iguais a *P* com qualquer caso de teste *T*. Caso isso ocorra, *p* é considerado vivo para posterior avaliação pelo desenvolvedor. Não havendo um caso de teste que diferencie *p* de *P*, e não sendo possível a geração de um novo conjunto *T* e, tendo a saída de *P* como correta, *p* é considerado equivalente a *P*, no sentido de que, ou *P* está correto, ou possui erros pouco prováveis de ocorrerem (DeMillo et al., 1978; Delamaro et al., 2007).

Define-se que um conjunto de casos de teste *T* é adequado para um programa *P* em relação a *P'*, se para cada programa *p* pertencente a *P'*, ou *p* é equivalente a *P* ou *p* difere de *P* em pelo menos um caso de teste (Delamaro et al., 2007). A determinação de equivalência é sempre realizada pelo testador, pois determinar se dois programas computam a mesma função é uma questão indecidível (Budd, 1981).

Uma adequação dos casos de teste utilizados em *P* e *P'* é obtida através do escore de mutação. O escore de mutação varia entre 0 e 1 e fornece uma medida objetiva de quanto o conjunto de casos de teste analisado é adequado (Delamaro et al., 2007; DeMillo et al., 1978). Para um programa *P* e um conjunto de casos de teste *T*, o escore de mutação *S* é obtido da seguinte maneira (DeMillo et al., 1978), conforme mostra a Equação 2.1.

$$S(T) = \frac{Mm}{(Mg - Me)} \tag{2.1}$$

- *Mm* é o número de mutantes mortos;
- *Mg* é o número de mutantes gerados;
- *Me* é o número de mutantes equivalentes.

Quanto maior for o escore de mutação obtido para um conjunto de casos de teste, ou seja, quanto mais próximo de 1 for o resultado, mais confiável se torna esse conjunto.

2.3 TRABALHOS RELACIONADOS

Nesta seção são apresentados os trabalhos relacionados à pesquisa desta tese. Na Subseção 2.3.1 são descritos de forma resumida os trabalhos de destaque no contexto de avaliação e teste de ontologias. Na Subseção 2.3.2 é apresentada uma lista de defeitos para ontologias, que resultou da elaboração de uma revisão bibliográfica dos tipos de defeitos ocasionados durante a modelagem e desenvolvimento de ontologias. Estes defeitos serviram como base para a elaboração de um conjunto de operadores de mutação para ontologias OWL que são apresentados na Seção 3.4.

2.3.1 Avaliação e teste de ontologias

Conforme apresentado por Vrandečić (2010), a avaliação de ontologias é a tarefa de medir a qualidade de uma ontologia. A qualidade da ontologia pode estar associada a diferentes cenários, como identificação de erros e omissões que possam causar a incapacidade de uso de uma ontologia; definir um modo de avaliação dos resultados que auxilie o processo de construção de uma ontologia e possíveis refinamentos; e possibilitar verificar se as restrições e os requisitos da ontologia foram atendidos. Com relação ao teste de ontologias, Blomqvist et al. (2012) descreve como sendo um tipo de avaliação focada em tarefas para avaliar os requisitos ontológicos, como um conjunto de questões de competência.

Neste contexto, seguindo as abordagens de Vrandečić (2010) e Blomqvist et al. (2012), aborda-se neste trabalho que o teste refere-se a um processo de executar uma ontologia com o objetivo de revelar a presença de defeitos, contribuindo para aumentar a confiança de que a ontologia desempenha as funções especificadas e deste modo, auxiliar o processo de avaliação propondo métodos para medir a qualidade das ontologias. Sendo assim, o teste de ontologias tem um propósito semelhante ao teste da engenharia de software, que é examinar o comportamento de uma ontologia por meio de sua execução.

O processo de avaliação de ontologias apresenta alguns desafios, devido à natureza declarativa das ontologias os desenvolvedores não podem apenas compilá-las e executá-las como na maioria dos softwares (Vrandečić, 2010). Diante disto, Vrandečić (2010) apresenta quatro níveis diferentes em que a avaliação de ontologias pode ser conduzida:

1. as ontologias podem ser avaliadas por si mesmas, ou seja, verificar se a ontologia satisfaz os requisitos definidos em sua construção. Como por exemplo, aplicação de questões de competência e análise dos resultados;

2. as ontologias podem ser avaliadas com algum contexto, conforme proposto por Gangemi et al. (2006) e Maedche e Staab (2002), onde a ontologia é comparada a outro modelo construído de forma ideal, conhecido como Padrão Ouro (do inglês, *Gold Standard*);

3. as ontologias podem ser avaliadas dentro de uma aplicação. Conforme Brank et al. (2005), este tipo de avaliação é denominada como Avaliação de Ontologias Baseadas em Aplicação, onde as ontologias podem ser avaliadas conectando-as a um aplicativo e avaliando os resultados deste aplicativo; e

4. as ontologias podem ser avaliadas no contexto de uma aplicação e uma tarefa. Segundo Porzel e Malaka (2004), esta avaliação é definida como Avaliação de Ontologias Baseada em Tarefas, onde avalia-se a ontologia na execução de uma tarefa para a qual foi desenvolvida. Primeiramente escolhe-se uma tarefa a ser executada pela ontologia e em seguida um padrão ouro, que servirá como modelo com respostas adequadas para comparação com a ontologia avaliada.

De acordo com Vrandečić (2010), na avaliação de ontologias os defeitos são mais facilmente identificados nos níveis 1 e 2, dado que para uma ontologia ser avaliada dentro de uma aplicação, ela deve ter sido avaliada por si mesma e em algum contexto.

Para Obrst et al. (2007), uma ontologia pode ser avaliada por muitos critérios, através da complexidade e granularidade da cobertura de um domínio; através de casos de uso, cenários, requisitos, aplicativos e fontes de dados desenvolvidas para essa finalidade; e pela consistência e completude da linguagem de representação na qual a ontologia é modelada. Ainda segundo Obrst et al. (2007), a avaliação de ontologias inclui aspectos de validação e verificação, como questões estruturais, funcionais e de usabilidade. Nesse contexto, elaboramos uma revisão sistemática da literatura sobre avaliação de ontologias (Porn et al., 2016), onde foram identificados diversos trabalhos relacionados que se destacaram e fundamentaram o desenvolvimento desta tese.

Grüninger e Fox (1995) propuseram o conceito e uso de questões de competência para guiar o processo de avaliação e adequação de ontologias. Após a definição de questões de competência informais, essas questões são convertidas em questões formais utilizando uma terminologia de lógica de primeira ordem. Essas questões são utilizadas como axiomas para a análise da representação do conhecimento na avaliação da ontologia.

Em Poveda-Villalón et al. (2012) é apresentado um método para validação de ontologias utilizando uma ferramenta *web*. O objetivo desse trabalho é melhorar a qualidade da ontologia através da detecção automática de enganos que poderiam ocasionar erros de modelagem. Com base em um catálogo de enganos definidos em linguagem natural, esses enganos são transformados em uma linguagem formal ou procedimental e implementados na ferramenta. Porém, é possível a detecção de apenas 21 tipos de enganos que podem ser cometidos. Desse modo, diversos outros enganos relevantes, como a utilização incorreta do operador OWL *AllValuesFrom* ou a definição incorreta de domínios ou intervalos (do inglês, *range*) nas propriedades não são detectados.

Em Baumeister e Seipel (2005) é proposto avaliar ontologias com base nos erros taxonômicos apontados por Gómez-Pérez (1999). Utilizando uma representação XML própria, a qual os autores denominam como *FNQuery*, implementam regras na linguagem de programação *Prolog* com o objetivo de identificar esses erros. Também propõem a simulação de anomalias que não são identificadas por *reasoners*, através de regras *FNQuery*, para analisar o comportamento e consequentemente a existência desses erros na ontologia.

Já em Baumeister e Seipel (2010) é apresentado um método de verificação de ontologias no nível simbólico utilizando uma abordagem declarativa. É proposto um complemento às metodologias de desenvolvimento de ontologias existentes. São investigadas as implicações e problemas que surgem nas definições de regras lógicas com a utilização de descritores ontológicos. São avaliadas as relações de classes, como subclasses, complemento e disjunção e; características básicas de propriedades, como transitividade, simetria, intervalos, domínios e restrições de cardinalidade. Nesse trabalho não são considerados erros comuns que possam ser implementados devido à compreensão incorreta das implicações lógicas da linguagem OWL, conforme descrito em Corcho et al. (2009), Poveda-Villalón et al. (2014) e Rector et al. (2004).

Em Rieckhof et al. (2011) é proposto um método para validar ontologias considerando três critérios: consistência, completude e corretude. Para o critério de consistência, os autores propõem a aplicação de *reasoners* sobre as definições da ontologia. Já para os critérios de completude e corretude, propõem primeiramente a definição de restrições de integridade utilizando a linguagem SPARQL, e posteriormente o processamento dessas restrições por um *reasoner*. Esse trabalho não possibilita a identificação de defeitos que não estejam associados a inconsistências e não sejam detectados por *reasoners*.

No contexto de identificar erros de modelagem, Vrandečić e Gangemi (2006) propõem aplicar testes de unidade em ontologias, como na engenharia de software, através do uso de

questões de competência. Essas questões devem ser formalizadas em uma linguagem de consulta para ontologias, para que descrevam o tipo de conhecimento que a ontologia deva responder. Ao formalizar uma questão o resultado deve satisfazer as condições da ontologia, caso contrário, um problema foi encontrado.

Focando em erros cometidos no desenvolvimento de ontologias identificados na literatura, Qazi e Qadir (2010) propuseram o uso de algoritmos para a identificação de erros circulatórios na hierarquia de classes e propriedades, redundância de subclasses e subpropriedades, redundância em relações de disjunção e omissão do conhecimento disjunto. A desvantagem no uso desse método refere-se a baixa quantidade de erros que podem ser identificados.

No trabalho de García-Ramos et al. (2016), é apresentada uma ferramenta para automatizar a execução dos testes de instanciação, recuperação, realização, satisfação, classificação e consultas SPARQL. O usuário precisa especificar uma questão de competência para executar na ferramenta contra a ontologia em teste. Para analisar o resultado obtido, primeiramente deve-se especificar o resultado esperado. Essa tarefa torna-se difícil de ser realizada quando são testadas ontologias já disponibilizadas para uso, devido ao desconhecimento dos conceitos representados na ontologia pelo testador. Outra dificuldade encontrada é quando a ontologia possui uma representação muito grande, com centenas ou milhares de conceitos, o que torna difícil a especificação do resultado esperado.

Já em Blomqvist et al. (2012), é proposto encontrar erros através da verificação de inferências na ontologia. Questões de competência são formuladas e executadas, os resultados obtidos são analisados e em seguida simulações de erros de inconsistência são inseridos. A simulação de erros assemelha-se ao teste de mutação. Porém, não são especificados operadores de mutação para essas simulações. Por outro lado, erros de inconsistência são identificados por *reasoners* em ferramentas de edição e construção de ontologias.

Em Corcho et al. (2004) é proposta uma ferramenta para detectar somente os erros circulatórios, de partição e de redundância, através da análise do grafo da ontologia. Uma vantagem dessa ferramenta é que ela se aplica tanto na linguagem OWL como em RDF(S) e DAML+OIL. Porém, baseia-se na identificação de um número limitado de erros.

Em se tratando do teste de mutação para ontologias OWL, alguns trabalhos podem ser observados em Lee et al. (2008), Porn e Peres (2014), Bartolini (2016) e Porn e Peres (2017).

Em Lee et al. (2008) são apresentados e aplicados operadores de mutação para o teste de mutação para OWL-S, um padrão de linguagem baseado em XML para especificar fluxos de trabalho e integração semântica entre serviços web. Foram analisados padrões de defeitos específicos de OWL-S e proposto um método de análise de mutação baseado em ontologias para simulação e teste de serviços web.

Em Porn e Peres (2014) aplicamos o teste de mutação exclusivamente para ontologias OWL. Foram definidos 19 operadores de mutação sintáticos para classes e propriedades, com o objetivo de encontrar defeitos em ontologias OWL. Porém, nesse trabalho não são realizadas mutações nos axiomas lógico-descritivos da ontologia, não é definida uma representação formal para a elaboração de um conjunto de dados de teste e também não é apresentado um método para a geração automática dos dados de teste. Por outro lado, os operadores de mutação apresentados neste trabalho não abrangem toda a especificação da linguagem OWL, o que possibilita que alguns tipos de defeitos não sejam possíveis de serem detectados com o método de teste proposto. Isso também ocorre pelo fato de os operadores de mutação não estarem associados aos possíveis defeitos que podem ser ocasionados ao construir uma ontologia.

Um processo similar é apresentado por Bartolini (2016). Neste trabalho são apresentados 22 operadores de mutação para o teste de mutação semântico para ontologias OWL. Porém, 16 operadores são similares aos operadores que propomos em Porn e Peres (2014). Outros

2 operadores sintáticos propostos neste trabalho são aplicados em estruturas de anotações da linguagem OWL que não causam impacto lógico na ontologia e portanto são desconsiderados. Dos 4 operadores restantes, 3 são aplicados na alteração da classe para a qual um indivíduo pertence, não sendo utilizados nesta tese por ser uma mutação que não se enquadra a um dos possíveis tipos de defeitos ocasionados em ontologias, e 1 operador é aplicado na definição estrutural de uma propriedade, também não relacionado a um tipo de defeito que possa ser ocasionado. Os 22 operadores apresentados neste trabalho não são possíveis de serem aplicados em axiomas lógico-descritivos para avaliar conceitos semânticos OWL. Os resultados apresentados não fazem referência a possíveis ocorrências de defeitos nas ontologias testadas, não demonstram o escore de mutação alcançado para avaliar a qualidade dos dados de teste, não é apresentado o formato dos dados aplicados, e também não é proposto um método para a geração desses dados.

Já em Porn e Peres (2017) propomos a definição de operadores de mutação para expressões lógicas definidas em Lógica Descritiva OWL e a geração automática de dados de teste. Foram reaproveitados e adaptados 8 operadores de Porn e Peres (2014) para serem aplicados em axiomas que representam restrições lógicas em classes, propriedades e indivíduos das ontologias OWL. Foi realizado um estudo de caso demonstrando a aplicabilidade desses operadores e a qualidade dos dados de teste gerados. Esse trabalho apresentou resultados satisfatórios para a definição do conjunto de operadores de mutação que fundamentam o teste de mutação para ontologias OWL apresentado nesta pesquisa.

Assim, nesta tese, são propostos operadores de mutação para classes, propriedades de objetos e de tipos de dados, indíviduos e axiomas, para aplicar o teste de mutação para ontologias OWL. Também é apresentada uma ferramenta para automatizar a geração dos mutantes e dos dados de teste. Os operadores de mutação apresentados nesta pesquisa são definidos para realizar alterações sintáticas nas ontologias OWL. Porém, esses operadores possibilitam a revelação e análise tanto de defeitos sintáticos como semânticos, causados pela má definição de axiomas.

2.3.2 Tipos de defeitos em ontologias

Conforme Delamaro et al. (2007) e a interpretação dos conceitos do padrão IEEE 610.12 (1990), na engenharia de software os defeitos em programas caracterizam-se por passos, processos ou dados incorretos e são ocasionados devido a uma ação humana denominada engano, de modo que esta definição pode também ser aplicada para ontologias. Os conceitos de engano e defeito são estáticos, pois estão associados a uma determinada ontologia e não dependem de uma execução em particular.

No contexto de ontologias, de acordo com Copeland et al. (2013), um defeito (do inglês, *fault*) em uma ontologia O pode ser considerado como um desvio entre o comportamento esperado e o comportamento executado. Segundo Copeland (2016) os defeitos em ontologias podem ser divididos em defeitos lógicos e não-lógicos, de modo que os defeitos lógicos tratam diretamente dos aspectos orientados pela lógica da ontologia, especificamente no conhecimento afirmado e implícito. Esses defeitos podem ser caracterizados como defeitos funcionais, pois tratam diretamente do aspecto lógico que a ontologia deve fornecer. Os defeitos não-lógicos tratam de como o conhecimento é modelado e documentado na ontologia. Assim, as decisões de modelagem em relação aos padrões, a especificidade das definições de conceitos, a informação sobre a ontologia nos axiomas de anotação e o compromisso ontológico estão diretamente relacionados aos aspectos não-lógicos ou não funcionais da ontologia.

Para Corcho et al. (2009), a diferença entre o comportamento esperado e o comportamento executado de uma ontologia é abordado como antipadrões (do inglês, *anti-pattern*). Estes antipadrões representam um conjunto de padrões que são normalmente utilizados pelos especialistas do domínio para a implementação de ontologias OWL e geralmente causam inconsistências

que devem ser resolvidas para a correta execução da ontologia. Outra abordagem também é apresentada por Poveda-Villalón et al. (2010), que define como armadilhas (do inglês, *pitfalls*) as piores práticas comuns utilizadas para a construção de ontologias, que levam o desenvolvedor a implementar incorretamente os conceitos, os quais podem resultar em comportamentos inesperados da ontologia.

Conforme Copeland (2016), os defeitos em ontologias estão geralmente relacionados à falta de consistência de modelagem, à adesão insuficiente ao compromisso ontológico do domínio, aos erros no conhecimento definido e às conseqüências erradas ou não-lógicas do conhecimento afirmado. Os defeitos podem ser introduzidos por ação humana, através de decisões de modelagem, falta de conhecimento do domínio ou através do uso de ferramentas, como ferramentas de geração de conteúdo automático ou de junção de ontologias.

Nesse contexto, segundo Gómez-Pérez (2004); Rector et al. (2004); Poveda-Villalón et al. (2010) e Corcho et al. (2009), como não existe apenas uma única forma correta de modelar um domínio do conhecimento específico, diversos tipos de defeitos podem ser cometidos. A seguir, é apresentada uma lista de defeitos resultante de uma revisão bibliográfica dos tipos de defeitos que podem ocorrer durante a modelagem e desenvolvimento de ontologias.

Gómez-Pérez (2004) apresenta os principais defeitos que podem ocorrer ao modelar conceitos e ideias no contexto de ontologias e define-os em três classes: defeitos de inconsistência, defeitos de incompletude e defeitos de redundância. Outra classificação dos principais defeitos que podem ocorrer ao modelar conceitos e ideias no contexto de ontologias pode ser encontrada em Rector et al. (2004). Poveda-Villalón et al. (2010), visando avaliar a corretude das ontologias quanto a sua qualidade, analisaram o desenvolvimento de um conjunto de 26 ontologias com a finalidade de catalogar erros comuns na fase de desevolvimento e objetivando eliminar essas práticas catalogaram 24 defeitos ocorridos. Corcho et al. (2009), estabeleceram uma lista de defeitos ocasionados por desenvolvedores de ontologias ou especialistas da área do conhecimento, baseando-se em uma ontologia OWL desenvolvida pelo *Spanish National Geographic Institute* - IGN-E, denominada *HydrOntology*, com aproximadamente 150 classes, 34 propriedades de objetos, 66 propriedades de dados e 256 axiomas, sendo definida para representar informações hidrográficas a nível regional, nacional e local. Esses defeitos são definidos por Corcho et al. (2009) como antipadrões, devido à maioria dos defeitos serem causados por padrões adotados por especialistas da área que normalmente resultam em inconsistências. Foram catalogados 11 defeitos definidos como antipadrões lógicos, 6 defeitos definidos como antipadrões não-lógicos e 4 defeitos definidos como diretrizes, totalizando ao todo 21 defeitos que em sua maioria equivalem-se aos problemas apresentados por Poveda-Villalón et al. (2010).

A lista dos possíveis defeitos ocorridos durante o desenvolvimento de ontologias é apresentada conforme segue:

1. **Defeitos de inconsistência (Gómez-Pérez, 2004):** esses defeitos geralmente ocorrem quando o desenvolvedor faz uma definição lógica na ontologia que cause uma inconsistência em sua estrutura durante a sua classificação. Eles são classificados em três tipos: defeitos circulatórios, defeitos de partição e defeitos de inconsistência semântica.

   1.a) **Defeito circulatório:** ocorre quando uma classe é definida como uma subclasse ou superclasse de si mesma em qualquer nível da hierarquia da ontologia.
   Exemplo: Uma classe C é definida como subclasse da classe B, que é definida como subclasse da classe A, onde esta é definida como subclasse da classe C.

   1.b) **Defeito de partição:** uma ou várias instâncias pertencem a mais de uma subclasse na definição da divisão da superclasse, ou quando uma classe é definida como

subclasse de outras duas ou mais subclasses na divisão da superclasse.
Exemplo: uma classe *Pessoa* é particionada nas subclasses *Homem* e *Mulher*, e define-se uma classe *Paciente* como subclasse tanto de *Homem* quanto de *Mulher*.

1.c) **Defeito de inconsistência semântica:** ocorre quando um conceito é classificado como uma subclasse de um conceito ao qual ele não pertence, sendo portanto, definida uma classificação semântica incorretamente.
Exemplo: classificar o conceito *Animal* como subclasse de *Pessoa* ou classificar um indivíduo da classe *Animal* como instância de *Pessoa*.

2. **Defeitos de incompletude (Gómez-Pérez, 2004):** esses defeitos ocorrem na maioria das vezes quando se define um relacionamento entre subclasses, entretanto se omite o conhecimento disjunto entre essas classes, sendo portanto, caracterizados como defeitos de classificação incompleta e defeitos de omissão do conhecimento disjunto.

2.a) **Defeito de classificação incompleta:** omitem-se alguns dos conceitos presentes no domínio do conhecimento durante a classificação dos subconceitos.
Exemplo: definir somente as classes *Criança* e *Adulto* como subclasses da classe *Pessoa* e omitir outras possibilidades necessárias como *Adolescente* ou *Idoso*.

2.b) **Defeito de omissão do conhecimento disjunto:** causado pela ausência de axiomas ou informações sobre a classificação de um conceito, através da omissão da divisão de subclasses disjuntas.
Exemplo: definir as classes *Homem* e *Mulher* como subclasses de *Pessoa* e omitir que ambas as subclasses são disjuntas. Ao omitir-se a disjunção, permite-se que um indivíduo seja instanciado como *Homem* e *Mulher* simultaneamente.

3. **Defeitos de redundância (Gómez-Pérez, 2004):** é um tipo de defeito que geralmente ocorre quando expressões já definidas explicitamente na ontologia são redefinidas ou são inferidas a partir de outras definições. Estes defeitos são abordados como defeito de redundância gramatical e defeito de definição formal idêntica de classes e instâncias.

3.a) **Defeito de redundância gramatical:** ocorre quando existe mais de uma definição explícita de qualquer relação da hierarquia da ontologia, tal como redundância de relação de subclasse ou redundância de relação de instância.
Exemplo: define-se em uma expressão lógica que a classe *Criança* é equivalente à classe *Pessoa* e que satisfaz uma condição imposta por uma propriedade de tipos de dados de que cada instância possua menos de 13 anos de idade. A mesma definição é imposta para a classe *Adolescente*, porém com a condição de que cada instância possua idade maior de 12 e menor de 18 anos. Além dessas duas definições, define-se ainda que a classe *Pessoa* é equivalente às classes *Criança* e *Adolescente*.

3.b) **Defeito de definição formal idêntica de classes e instâncias:** ocorre quando existem duas ou mais classes ou instâncias na ontologia com a mesma definição formal, sendo o nome a única diferença entre as subclasses e instâncias.
Exemplo: definir os conceitos *Jovem* e *Adolescente* com a expressão lógica de que são equivalentes à classe *Pessoa* e que satisfazem uma condição de uma propriedade de tipos de dados onde cada instância possui mais de 12 anos e menos de 18.

4. **Assumir características implícitas no nome dos componentes (Rector et al., 2004):** ocorre quando as definições lógicas dos componentes da ontologia (classes, propriedades ou instâncias) não estão disponíveis automaticamente para o classificador.
   Exemplo: não definir que os conceitos *Homem* e *Mulher* são disjuntos na ontologia, possibilita instanciar objetos para ambas as classes, pois a informação implícita no nome não é disponível para o classificador como a definição explícita da disjunção.

5. **Uso equivocado de qualificadores de restrições (Rector et al., 2004):** ocorre quando os operadores de restrição *allValuesFrom* - $\forall$ (universal) e *someValuesFrom* - $\exists$ (existencial) são utilizados incorretamente ou equivocadamente na definição de axiomas.
   Exemplo: definir a classe *Consulta* com a condição de que esta classe pode ter pessoas envolvidas que podem ser tanto instâncias da classe *Médico* quanto da classe *Paciente*, através das expressões lógicas "*someValuesFrom Médico*" e "*someValuesFrom Paciente*", mas se omite o fato de que nenhum outro tipo de instância pode estar associada à classe *Consulta*, o que não impede que também possam participar desta associação instâncias da classe definida como *Enfermeiro*. Devido a isso, a restrição *"allValuesFrom (Médico or Paciente)"* é necessária para completar a definição da classe *Consulta*.

6. **Suposição de mundo aberto (Rector et al., 2004):** ocorre quando se representa um conceito na ontologia e omite-se uma definição lógica para esse conceito com a intenção de que a ontologia interprete que esta definição não corresponde a este conceito por ela não ter sido definida. Isto ocorre porque, ao contrário da definição de sistemas *"Closed-World Assumption"* (como por exemplo os bancos de dados relacionais) que utilizam a negação como falha para presumir que algo está ausente quando não encontram, os classificadores de ontologias são do tipo *"Open-World Assumption"*, possibilitando usar a negação como insatisfatória, isto é, dizer que algo é falso apenas se puder ser provado.
   Exemplo: definir uma classe *MédicoClínicoGeral* com uma restrição de equivalência de que são médicos que não possuem relação com instâncias da classe *Especialidade*. Ao instanciar um indivíduo para essa classe, é necessário dizer explicitamente que esse indivíduo não possui qualquer especialidade.

7. **Domínio e intervalo de propriedades (Rector et al., 2004):** ocorre ao definir incorretamente as propriedades ou restrições de equivalência em uma classe. Diferentemente de outras linguagens de domínio onde erros ocorrem se as propriedades são definidas incorretamente, em ontologias a definição incorreta de propriedades pode tornar uma classe insatisfatória ou submetê-la a outra classe inesperadamente.
   Exemplo: considerando uma classe definida como *Jovem*, sendo esta uma subclasse da classe *Pessoa*, definida com um critério de equivalência de que são todos os indivíduos com a propriedade de dados *hasIdade* com valor menor a 18 para representar pessoas com menos de 18 anos. Se esta propriedade não tiver um domínio específico a ser aplicado (no caso a classe *Pessoa*), a propriedade *hasIdade* pode ser atribuída a qualquer tipo de indivíduo, de outras classes, e o classificador pode indicar como pertencente à classe *Jovem* não apenas instâncias das classes *Homem* ou *Mulher*, mas também qualquer tipo de indivíduo, como de uma classe definida como *Profissão*.

8. **Utilização incorreta dos operandos AND e OR (Rector et al., 2004):** aplicar incorretamente os operandos *AND* e *OR* em uma definição lógica, de modo que sejam apresentados resultados incompletos na aplicação de um classificador na ontologia.

Exemplo: definir as classes *Homem* e *Mulher* como disjuntas e, ao tentar localizar todas as pessoas do sexo masculino e todas as pessoas do sexo feminino utilizando o operando *"OR"*, as instâncias de ambas as classes serão localizadas, ao contrário do operando *"AND"*, que não localizaria nenhuma das instâncias por não haver instâncias que correspondessem a ambas as classes simultaneamente.

9. **Criação de elementos polissêmicos (Poveda-Villalón et al., 2010):** elementos cujo nome tem significados diferentes que são incluídos na ontologia para representar mais de uma ideia conceitual.
   Exemplo: definir uma classe *AgenteDeSaúde* para representar tanto indivíduos da classe *Médico* como da classe *Enfermeiro*, sendo que estas classes representam conceitos diferentes uma da outra.

10. **Criação de classes sinônimas (Poveda-Villalón et al., 2010):** ocorre ao criar duas ou mais classes com identificadores sinônimos e definí-las como equivalentes. Este defeito também é mencionado por Gómez-Pérez (2004) como *Definição formal idêntica*.
    Exemplo: representar em uma ontologia as classes *Carros*, *Veículos* e *Automóveis*, definidas como equivalentes, sendo os indivíduos os mesmos para cada classe.

11. **Não definir elementos primitivos (Poveda-Villalón et al., 2010):** refere-se ao fato de ao invés de representar uma subclasse com a propriedade *"subclassOf"*, membros de uma classe com *"instanceOf"* ou a igualdade entre instâncias com o construtor *"sameAs"*, é definida uma propriedade de objetos do tipo *"is"*, para representar a associação entre os componentes.
    Exemplo: representar as classes *Carro* e *Automóvel* utilizando a propriedade *"is"* com a definição da expressão *"Carro is Automóvel"*, ao invés de definir a classe *Carro* como subclasse de *Automóvel*.

12. **Criação de elementos desconexos (Poveda-Villalón et al., 2010):** ocorre ao definir componentes na ontologia (classes, propriedades ou indivíduos) sem relação com o resto da ontologia.
    Exemplo: criar o relacionamento *"membrosDoTime"* e não definir a classe que representa as equipes, assim, a relação criada é isolada na ontologia.

13. **Definição incorreta de propriedades inversas (Poveda-Villalón et al., 2010):** ocorre ao definir incorretamente uma propriedade como inversa de outra.
    Exemplo: representar em uma ontologia que *"algo é vendido"* ou *"algo é comprado"* através das propriedades *"vendidoEm"* e *"compradoEm"*, e definí-las como inversas, sendo que estas propriedades representam conceitos diferentes.

14. **Inclusão de ciclos na hierarquia (Poveda-Villalón et al., 2010):** incluir duas classes na hierarquia da ontologia para representar uma como superclasse ou subclasse da outra. Este defeito também é citado por Gómez-Pérez (2004) como *Defeito Circulatório*.
    Exemplo: uma classe "A" possui como subclasse uma classe "B" e, ao mesmo tempo essa classe "B" é definida como superclasse de "A".

15. **Criar uma classe para representar diferentes conceitos (Poveda-Villalón et al., 2010):** criar uma classe com um identificador que se refere para mais de um conceito.
    Exemplo: definir uma classe *"Produto_Serviço"* para representar indivíduos diferentes.

16. **Anotações incompletas (Poveda-Villalón et al., 2010):** ocorre pela falta de propriedades de anotações nos termos utilizados na ontologia. Este tipo de propriedade melhora a compreensão da ontologia e usabilidade do ponto de vista do utilizador.
Exemplo: definir a classe *Consulta* na ontologia e não especificar em linguagem natural através das propriedades de anotações o significado desta classe.

17. **Informações básicas incompletas (Poveda-Villalón et al., 2010):** não incluir informações necessárias na ontologia.
Exemplo: definir uma propriedade *"iniciaEm"* para representar o ponto inicial de uma rota, mas deixar de definir uma propriedade *"terminaEm"* para representar o seu fim.

18. **Falta de disjunção (Poveda-Villalón et al., 2010):** ocorre devido a falta da definição de disjunção entre classes ou propriedades da ontologia. Este defeito também é mencionado por Gómez-Pérez (2004) como *Omissão do Conhecimento Disjunto*.
Exemplo: para uma classe *Pessoa* que possui duas subclasses *Homem* e *Mulher*, e estas não estão definidas como disjuntas, é possível que um indivíduo seja instanciado para ambas as classes ao mesmo tempo.

19. **Falta de domínio ou intervalo em propriedades (Poveda-Villalón et al., 2010):** ocorre ao definir propriedades ou indivíduos na ontologia sem determinar seu domínio ou intervalo de dados. Este defeito também é mencionado por Rector et al. (2004) como *Domínio e Intervalo de Propriedades*.
Exemplo: definir a propriedade de objetos *escritoPor*, onde o domínio deveria ser a classe *Escritor* e o intervalo a classe *ObraLiterária*, mas um dos dois não é especificado.

20. **Falta de equivalência entre propriedades (Poveda-Villalón et al., 2010):** geralmente são ocasionados quando ocorre a importação de uma ontologia dentro de outra, e as classes ou propriedades que são repetidas não são definidas como equivalentes.
Exemplo: duas classes definidas como *Cidade* e *Município* de duas ontologias diferentes são definidas como equivalentes, mas as propriedades *hasMembro* e *has-Membro* não.

21. **Falta de propriedades inversas (Poveda-Villalón et al., 2010):** ocorre quando é omitida a definição de inversão entre duas propriedades.
Exemplo: não definir que as propriedades *maeDe* e *filhoDe* são inversas.

22. **Uso incorreto da restrição allValuesFrom (Poveda-Villalón et al., 2010):** esse defeito pode ocorrer em duas maneiras diferentes. No primeiro caso, a anomalia refere-se ao utilizar como qualificador padrão a restrição universal (*"allValuesFrom"*) quando deveria ser utilizada a restrição existencial (*"someValuesFrom"*), dando a falsa impressão ao desenvolvedor de que *allValuesFrom* implica *someValuesFrom*. No segundo caso, ocorre ao incluir a restrição *"allValuesFrom"* com a finalidade de impossibilitar novas adições para uma determinada propriedade. Este defeito também é mencionado por Rector et al. (2004) como *Uso Equivocado de Restrições*.
Exemplo: definir que a classe *Hospital* é composta por indivíduos das classes *Médicos* e *Enfermeiros*, impossibilitando adicionar indivíduos da classe *Pacientes*.

23. **Uso incorreto da definição "some not" (Poveda-Villalón et al., 2010):** confundir a representação de "some not".
Exemplo: definir que a classe *PizzaVegetariana* é equivalente a *qualquer pizza que não tem cobertura de carne ou cobertura de peixe*, isto possibilitará ser instanciado qualquer outro tipo de pizza como vegetariana, assim como uma com cobertura de chocolate.

24. **Uso incorreto de classes primitivas e definidas (Poveda-Villalón et al., 2010):** geralmente ocorre devido ao desconhecimento pelo desenvolvedor das ontologias serem *Open-World Assumption*, de modo que o fato de não definir uma propriedade para um indivíduo não determina que este indivíduo não faça parte desta propriedade.
Exemplo: não definir uma associação entre um indivíduo da classe *Médicos* e o indivíduo *cardiologista* através da propriedade *hasEspecialidade*, não garantirá que este indivíduo não seja um *cardiologista*.

25. **Alto grau de especialização (Poveda-Villalón et al., 2010):** ocorre devido a ontologia estar especializada de tal maneira que os "nós" finais da hierarquia não possam ser instânciados, devido ao fato de terem sido definidos como instâncias ao invés de classes.
Exemplo: definir uma classe *Lugar* e criar como suas instâncias os indivíduos *Barcelona*, *Madrid* e *Sevilla*, ao invés de definí-los como subclasses de *Lugar*.

26. **Intervalo de dados ou domínio limitado (Poveda-Villalón et al., 2010):** ocorre ao limitar o domínio ou intervalo de uma propriedade.
Exemplo: definir o domínio da propriedade *idiomaOficial* para a classe *Cidade* ao invés da classe *País*.

27. **Trocar os construtores intersecção e união (Poveda-Villalón et al., 2010):** ocorre pelo fato do desenvolvedor especificar o intervalo ou domínio de uma propriedade pela intersecção de várias classes, onde o correto seria definir pela união destas classes ou o contrário. Este defeito também é mencionado por Rector et al. (2004) como *Utilização incorreta dos operandos AND e OR*.
Exemplo: definir a propriedade *aconteceEm* com domínio para a classe *JogosOlimpicos* e como intervalo a intersecção das classes *Cidade* e *Nação*, ao invés da união destas classes.

28. **Trocar rótulos por comentários (Poveda-Villalón et al., 2010):** ocorre devido o desenvolvedor trocar o conteúdo de um comentário de um objeto por um rótulo.
Exemplo: especificar um comentário de uma definição da ontologia no construtor *Label* e um termo que define uma classe no construtor *Comment*.

29. **Definir uma classe miscelânea (Poveda-Villalón et al., 2010):** ocorre ao criar na hierarquia de uma ontologia uma subclasse que contenha instâncias que não pertençam as demais subclasses, ao invés de definir estas instâncias como instâncias da superclasse.
Exemplo: definir a classe *Animais* com suas subclasses *Gato*, *Cachorro* e *Diversos*, onde as instâncias da classe *Diversos* não se enquadram em *Gato* ou *Cachorro*.

30. **Utilizar diferentes critérios de nomenclatura (Poveda-Villalón et al., 2010):** ocorre quando o desenvolvedor não padroniza o modo de nomear os elementos da ontologia.
Exemplo: definir as classes e demais conceitos na ontologia utilizando em alguns casos letras em maíusculo, mínusculo ou com a primeira letra em maíusculo.

31. **Utilizar elementos incorretamente (Poveda-Villalón et al., 2010):** quando um elemento (classe, propriedade ou indivíduo) que foi usado para modelar uma parte desta ontologia deveria ter sido modelado como um elemento diferente.
Exemplo: definir uma associação entre uma instância da classe *Carros* e as instâncias *Sim* ou *Nao* utilizando uma propriedade de tipos de dados definida como *isEcologico*, onde deveria ser criado o atributo *isEcologico* cujo intervalo seria *boolean*.

32. **Definir um elemento recursivamente (Poveda-Villalón et al., 2010):** ocorre devido um elemento da ontologia ter sido usado em sua própria definição.
Exemplo: criar a propriedade *temCobertura* estabelecendo como intervalo a definição lógica: *pizzas que têm, pelo menos, um valor para a propriedade temCobertura*.

33. **Antipadrões lógicos (Corcho et al., 2009):** São defeitos ocasionados pelo desenvolvedor e que são detectados em sua maioria pelos classificadores de Lógica Descritiva.

33.a) **AndIsOr (AIO)**: ocorre quando o desenvolvedor da ontologia aplica equivocadamente os operandos lógicos "*and*" e "*or*" em uma definição lógica para classificar ou recuperar informação, utilizando um dos operandos quando o correto seria aplicar o outro. É um erro de modelagem comum que aparece devido ao fato de que no uso linguístico comum, "e" e "ou" não correspondem consistentemente à conjunção e disjunção lógicas respectivamente. Este defeito também é mencionado por Rector et al. (2004) como *Utilização incorreta dos operandos AND e OR* e, por Poveda-Villalón et al. (2010) como *Trocar os construtores intersecção e união*.
Exemplo: $C1 \subseteq (\exists R.C2 \cap C3)$, $Disj(C2,C3)$

33.b) **EquivalenceIsDifference (EID):** surge quando uma classe *C1* é definida como equivalente a classe *C2* ($C1 \equiv C2$) quando deveria ser definida como subclasse, pois *C2* diferencia-se de *C1* por apresentar mais informações ($Disj(C1,C2)$). Neste caso, o que realmente deve ser expresso é que *C1* é um subconjunto de *C2* ($C1 \subseteq C2$).
Exemplo: $C1 \equiv C2$, $Disj(C1,C2)$

33.c) **OnlynessIsLoneliness (OIL):** ocorre quando o desenvolvedor da ontologia cria uma restrição universal *R* para dizer que a classe *C1* só pode estar vinculada com a classe *C2*. Em seguida, uma nova restrição universal *R* é adicionada dizendo que C1 só pode estar vinculada com a classe *C3*, onde as classes *C2* e *C3* são disjuntas. Em geral, isso significa que o desenvolvedor esqueceu o axioma anterior.
Exemplo: $C1 \subseteq \forall R.C2$, $C1 \subseteq \forall R.C3$, $Disj(C2,C3)$

33.d) **OnlynessIsLonelinessWithInheritance (OILWI):** esse defeito é uma especialização do defeito OIL. Ocorre quando o desenvolvedor da ontologia adiciona uma restrição universal *R* para a classe *C1*, sendo que já havia definido outra restrição universal *R* com a mesma propriedade para a classe pai de *C1*.
Exemplo: $C1 \subseteq C2$, $C1 \subseteq \forall R.C3$, $C2 \subseteq \forall R.C4$, $Disj(C3,C4)$

33.e) **OnlynessIsLonelinessWithPropertyInheritance (OILWPI):** também é uma especialização de OIL. O desenvolvedor interpreta de forma equivocada as relações de subpropriedades, interpretando-as como sendo semelhantes a uma relação *part-of* em OWL, porque $C1 \subseteq \forall R1.C2$, $R1 \subseteq R2 \models C1 \subseteq \forall R2.C2$.
Exemplo: $R1 \subseteq R2$, $C1 \subseteq \forall R1.C2$, $C1 \subseteq \forall R2.C3$, $Disj(C2,C3)$

33.f) **UniversalExistence (UE):** ocorre quando o desenvolvedor da ontologia adiciona uma restrição existencial *R* para um conceito, sendo que já existe uma restrição universal *R* para este mesmo conceito, ocasionando uma inconsistência.
Exemplo: $C1 \subseteq \forall R.C2$, $C1 \subseteq \exists R.C3$, $Disj(C2,C3)$

33.g) **UniversalExistenceWithInheritance1 (UEWI_1):** esse defeito é uma especialização do defeito UE. O desenvolvedor da ontologia adiciona uma restrição existencial/universal em um conceito, sendo que já havia uma restrição universal/existencial para uma classe pai desse conceito, levando a uma inconsistência.
Exemplo: $C1 \subseteq C2$, $C1 \subseteq \exists R.C3$, $C2 \subseteq \forall R.C4$, $Disj(C3,C4)$

33.h) **UniversalExistenceWithInheritance2 (UEWI_2):** esse defeito também é uma especialização do defeito UE, pois apresenta o mesmo formato do defeito UEWI_1, apenas alterando a sequência em que as restrições são adicionadas. Deste modo, o desenvolvedor da ontologia adiciona uma restrição universal/existencial em um conceito, sendo que já havia uma restrição existencial/universal para uma classe pai desse conceito, levando a uma inconsistência.
Exemplo: $C1 \subseteq C2$, $C1 \subseteq \forall R.C3$, $C2 \subseteq \exists R.C4$, $Disj(C3,C4)$

33.i) **UniversalExistenceWithPropertyInheritance1 (UEWPI_1):** esse defeito é similar ao defeito OILWPI, pois também ocorre quando o desenvolvedor da ontologia interpreta de forma equivocada as relações de subpropriedades, interpretando-as como sendo semelhantes a uma relação *part-of* em OWL, porque $C1 \subseteq \exists R1.C2$, $R1 \subseteq R2 \models C1 \subseteq \exists R2.C2$. Porém, é uma especialização do defeito UE por tratar-se de interpretação equivocada de restrições.
Exemplo: $R1 \subseteq R2$, $C1 \subseteq \exists R1.C2$, $C1 \subseteq \forall R2.C3$, $Disj(C2,C3)$

33.j) **UniversalExistenceWithInverseProperty (UEWIP):** esse defeito ocorre quando o desenvolvedor da ontologia adiciona restrições sobre as classes *C2* e *C1* utilizando uma propriedade *R* e sua propriedade inversa R^{-1}, quando as classes *C2* e *C1* são definidas como disjuntas. Este defeito também é uma especialização do defeito UE, porque $C2 \subseteq \exists R^{-1}.C1 \models C1.1 \subseteq \exists R.C2$, $C1.1 \subseteq C1$.
Exemplo: $C2 \subseteq \exists R^{-1}.C1$, $C1 \subseteq \forall R.C3$, $Disj(C2,C3)$

33.k) **SumOfSomIsNeverEqualToOne (SOSINETO):** ocorre quando o desenvolvedor da ontologia adiciona uma nova restrição existencial *R* para um conceito, sendo que já existem restrições existencial e de cardinalidade para o mesmo conceito.
Exemplo: $C1 \subseteq \exists R.C2$, $C1 \subseteq \exists R.C3$, $C1 \leq 1R.T$, $Disj(C2,C3)$

34. **Antipadrões não-lógicos (Corcho et al., 2009):** representam possíveis erros de modelagem que não são detectados pelos classificadores DL.

34.a) **SynonymeOfEquivalence (SOE):** ocorre quando o desenvolvedor define que um conceito *C1* é equivalente a *C2* para representar uma relação terminológica sinônima, onde geralmente um dos conceitos não é utilizado em nenhum lugar na ontologia.
Exemplo: $C1 \equiv C2$

34.b) **SumOfSom (SOS):** ocorre quando o desenvolvedor adiciona uma nova restrição existencial, quando existe uma usando a mesma propriedade para o mesmo conceito. Embora em alguns casos isto seja correto, algumas vezes representa um erro.
Exemplo: $C1 \subseteq \exists R.C2$, $C1 \subseteq \exists R.C3$, $Disj(C2,C3)$

34.c) **SumOfSomWithInheritage (SOSWI):** é uma especialização do defeito SOS e ocorre quando o desenvolvedor adiciona uma nova restrição existencial em um conceito, sendo que já existe uma com a mesma propriedade para a classe pai.
Exemplo: $C1 \subseteq C2$, $C1 \subseteq \exists R.C3$, $C2 \subseteq \exists R.C4$, $Disj(C3,C4)$

34.d) **SumOfSomWithPropertyInheritance (SOSWPI):** similar aos defeitos OILWPI e UEWPI. O desenvolvedor interpreta de forma equivocada as relações de subpropriedades, interpretando-as como sendo semelhantes a uma relação *part-of* em OWL, porque $C1 \subseteq \exists R1.C2$, $R1 \subseteq R2 \models C1 \subseteq \exists R2.C2$. É uma especialização do defeito SOS por tratar-se do uso equivocado de herança de propriedades.
Exemplo: $R1 \subseteq R2$, $C1 \subseteq \exists R1.C2$, $C1 \subseteq \forall R2.C3$, $Disj(C2,C3)$

34.e) **SumOfSomWithInverseProperty (SOSWIP):** ocorre quando o desenvolvedor da ontologia cria duas restrições existenciais usando uma propriedade R e sua propriedade inversa R^{-1}. Esse defeito também é uma especialização do defeito SOS, considerando-se que $C2 \subseteq \exists R^{-1}.C1 \models C1.1 \subseteq C1$, $C1.1 \subseteq \exists R.C2$.
Exemplo: $C2 \subseteq \exists R^{-1}.C1$, $C1 \subseteq \exists R.C3$, $Disj(C2,C3)$

34.f) **SomeMeansAtLeastOne (SMALO):** ocorre quando o desenvolvedor da ontologia define uma restrição de cardinalidade supérflua, quando já existe uma restrição existencial que já indica que a restrição de cardinalidade é no mínimo igual a 1.
Exemplo: $C1 \subseteq \exists R.C2$, $C1 \leq 1R.T$

35. **Diretrizes (Corcho et al., 2009):** representam expressões complexas para definir os componentes da ontologia. Estão corretas de um ponto de vista lógico, porém poderiam ser utilizadas alternativas mais simples para representar o mesmo conhecimento.

35.a) **DisjointnessOfComplement (DOC):** ocorre quando o desenvolvedor da ontologia ao invés de representar dois conceitos $C1$ e $C2$ que não podem compartilhar instâncias definindo-os como disjuntos, utiliza o operador de negação *not*.
Exemplo: $C1 \equiv not\ C2$

35.b) **Domain&CardinalityConstraints (DCC):** esse defeito ocorre quando o desenvolvedor da ontologia define em conjunto uma restrição existencial com uma restrição de cardinalidade, uma vez que restrições existenciais contém restrição de carinalidade, dado que $C1 \subseteq \exists R.C2 \models C1$ ($\leq 1R.C2$). Neste caso, o que pode ocorrer é que seja necessário uma restrição universal com a definição de cardinalidade.
Exemplo: $C1 \subseteq \exists R.C2$, $C1 \subseteq (\leq 2R.T)$

35.c) **GroupAxioms (GA):** ocorre quando o desenvolvedor da ontologia não agrupa todas as restrições que utilizam a mesma propriedade R definidas em um conceito.
Exemplo: $C1 \subseteq \forall R.C2$, $C1 \subseteq (\leq 2R.T)$

35.d) **MinIsZero (MIZ):** ocorre quando o desenvolvedor da ontologia define que o conceito $C1$ é o domínio da propriedade R utilizando cardinalidade mínima 0 (zero). Este tipo de definição não tem impacto no modelo lógico e pode ser removida.
Exemplo: $C1 \subseteq (\leq 0R.T)$

Esses defeitos são baseados em sua maioria nas estruturas e modos de implementação das ontologias, sendo considerados para a definição dos operadores de mutação em conformidade com a sintaxe da linguagem OWL, para a aplicação do teste de mutação para ontologias OWL.

2.4 CONSIDERAÇÕES DO CAPÍTULO

Neste capítulo foi apresentada a representação de domínios específicos do conhecimento para a computação através de ontologias, as quais são objeto de estudo desta tese para a aplicação do teste de mutação, que tem como objetivo revelar defeitos ocasionados durante o seu desenvolvimento. Na Subseção 2.1.1 foram apresentados brevemente conceitos e definições de lógica descritiva para auxiliar a compreensão do desenvolvimento das ontologias, dado que a representação de uma ontologia é definida pelo mapeamento dessa lógica na linguagem em que a ontologia está sendo construída. Neste contexto, na Subseção 2.1.2 foi apresentada a definição de ontologias OWL para auxiliar na compreensão da representação dessas ontologias em classes, propriedades e indivíduos e, na Subseção 2.1.3 foi abordada a linguagem OWL para a construção

de ontologias, com o propósito de compreender o mapeamento da lógica descritiva na linguagem OWL, sendo inicialmente mostrado no Quadro 2.3 o mapeamento entre os principais elementos dessa lógica e seus respectivos construtores OWL. Na Subseção 2.1.4 apresentou-se as questões de competência que representam axiomas que são usados neste trabalho como dados de teste para a aplicação do teste de mutação.

Na Seção 2.2 foi abordada uma breve descrição da técnica de teste de mutação para programas, a qual é adaptada nesta tese para o teste de ontologias OWL. Já na Subseção 2.3.1 foi apresentado um levantamento dos principais métodos propostos na literatura para a avaliação de ontologias. Estes métodos têm como foco avaliar a corretude das ontologias, assim como detectar defeitos que ocorram no momento do desenvolvimento. Na Subseção 2.3.2 foi apresentada uma lista dos possíveis defeitos ocasionados no desenvolvimento de ontologias para apoiar a criação de novos operadores de mutação e a análise dos já existentes.

Visando uma contribuição para o teste de ontologias, no Capítulo 3 é abordada a descrição conceitual da proposta de teste de mutação para ontologias OWL desta tese.

3 CONTRIBUIÇÕES CONCEITUAIS PARA O TESTE DE MUTAÇÃO DE ONTOLOGIAS

Neste capítulo são apresentadas as contribuições desta tese referentes aos resultados dos itens (i), (ii) e (iii) apresentados na Seção 1.4. O item (i) que aborda a definição dos termos engano, defeito, erro e falha no contexto de ontologias para o teste de mutação é relatado na Seção 3.1; o item (ii) referente à caracterização de dados e casos de teste para a execução do teste de mutação para ontologias é abordado na Seção 3.2 e, na Seção 3.3 é apresentada de forma resumida a caracterização do teste de mutação para ontologias OWL com base na realização de estudos iniciais (Porn, 2014); e o item (iii) apresenta na Seção 3.4 a definição de um conjunto de operadores de mutação que abordem os termos da linguagem OWL e sejam compatíveis com a lista de defeitos apresentada na Subseção 2.3.2.

3.1 ENGANO, DEFEITO, ERRO E FALHA

Segundo Delamaro et al. (2007) e com base na interpretação dos conceitos de engano, defeito, erro e falha do padrão IEEE 610.12 (1990), a existência de um defeito em um programa leva à ocorrência de um erro, desde que o defeito seja executado. O erro caracteriza-se por um estado inconsistente ou inesperado devido à execução de um defeito. Este estado inconsistente ou inesperado pode ocasionar uma falha, fazendo com que o resultado produzido pela execução do programa seja diferente do resultado esperado. Neste trabalho, seguindo a abordagem de Delamaro et al. (2007) e a terminologia dos termos engano, defeito, erro e falha do padrão IEEE 610.12 (1990), esses termos são definidos no contexto de ontologias para a aplicação do teste de mutação conforme segue:

- **engano (do inglês, *mistake*):** é a ação humana que interpreta de maneira equivocada um conceito específico do domínio do conhecimento, produzindo um defeito que acarrete em um erro e possivelmente ocasione falhas indesejadas;
- **defeito (do inglês, *fault*):** é a representação incorreta de um conceito na ontologia, através da representação equivocada de algum elemento da sua estrutura de classes, propriedades, indivíduos ou axiomas lógicos. Um defeito pode acontecer devido à ocorrência de um engano cometido pelo desenvolvedor ou especialista da área do domínio do conhecimento abordado;
- **erro (do inglês, *error*):** é um estado inconsistente ou inesperado da ontologia em memória (interno ao sistema computacional) após a aplicação de uma consulta. Um erro ocorre devido a existência e execução de um defeito;
- **falha (do inglês, *failure*):** é a apresentação (externa ao sistema computacional) de um resultado diferente do resultado esperado após a aplicação de uma conulta na ontologia. Uma falha acontece devido a ocorrência de um erro.

Diante das definições de engano, defeito, erro e falha para ontologias, a Figura 3.1 apresenta um exemplo de uma estrutura de classes e propriedades de uma ontologia, onde o desenvolvedor comete um engano ao especificar o domínio de uma propriedade e utilizá-la para representar diferentes conceitos, levando a existência de um defeito na ontologia, que ao ser processado por uma consulta, consequentemente ocasionará um erro e levará a uma falha.

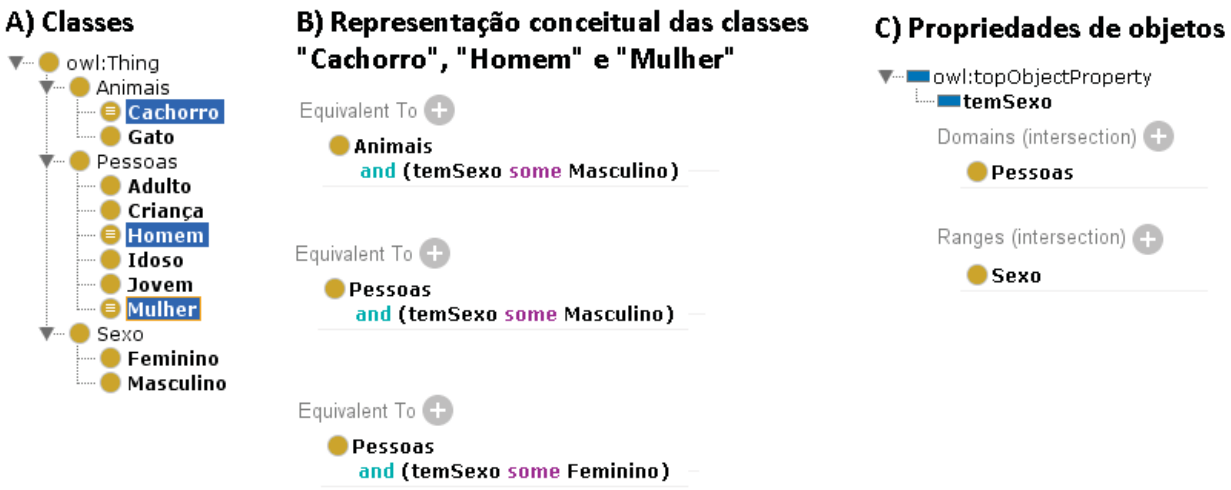


Figura 3.1: Exemplo de ocorrência de engano, defeito, erro e falha em uma ontologia (autor, 2019)

Na Figura 3.1, o desenvolvedor cometeu um engano ao definir a classe *Pessoas* como domínio da propriedade de objetos *temSexo* e em seguida representar as classes *Cachorro*, *Homem* e *Mulher* com essa mesma propriedade. Esse engano leva a existência de um defeito, dado que outros conceitos também necessitam serem representados pela propriedade *temSexo* e não somente *Pessoas*. Esse defeito ocasiona um estado inesperado da ontologia, apresentando um erro em sua classificação. Diante desse erro, ao realizar uma simples consulta, buscando por todos os objetos associados pela propriedade *temSexo* à classe *Masculino*, a ontologia apresentará uma falha ao retornar *Cachorro* como subclasse de *Pessoas*.

Na Subseção 2.3.2, Corcho et al. (2009) e Poveda-Villalón et al. (2010) apresentam as definições de antipadrões e armadilhas como representações anômalas que podem levar a ontologia a um estado inesperado. Diante desta afirmação e de acordo com a definição de defeito apresentada acima, estas anomalias podem ocasionar um erro levando a ontologia à uma falha. Assim, considera-se nesta tese que armadilhas e antipadrões também são tipos de defeitos.

3.2 DADOS DE TESTE E CASOS DE TESTE

Para o teste de mutação de aplicações no contexto de bancos de dados relacionais, Cabeça et al. (2009) e Tuya et al. (2007) consideram como dados de teste os parâmetros de entrada de uma consulta SQL, visto que o foco das mutações são as expressões SQL e não o modelo lógico ou físico do banco de dados. Analogamente, para o teste de ontologias OWL, propõe-se neste trabalho que os dados de teste sejam as consultas realizadas na ontologia, uma vez que se pretende testar o modelo conceitual da ontologia e não cada axioma lógico descritivo. Para Grüninger e Fox (1995), bons dados de teste para ontologias podem ser elaborados através de questões de competência - CQ, que descrevem os tipos de conhecimentos que a ontologia retorna ao responder estas questões. De acordo com Obrst et al. (2007), as CQs podem ser consideradas como uma coleção de dados de testes, fornecendo valores durante as atividades de análise e validação da ontologia. Já para García-Ramos et al. (2016), se por um lado tais CQs são o núcleo da especificação funcional da ontologia, por outro lado, são um conjunto de dados de teste que a ontologia deve transpor. Conforme Vrandečić e Gangemi (2006), estas questões devem ser formalizadas em uma linguagem de consulta, para que seja possível testar automaticamente se a ontologia relaciona os estados requeridos com estas questões.

De acordo com estas afirmações e com base nas definições para o teste de mutação para aplicações de bancos de dados relacionais, proposto por Cabeça et al. (2009) e Tuya et al. (2007), a Definição I apresenta a definição de dado de teste para o teste de mutação para ontologias OWL, onde as CQs estruturadas em lógica descritiva correspondem aos parâmetros de entrada para a composição de um conjunto de dados de teste *T*.

- **Definição I:** *Dado de teste* para ser aplicado em uma ontologia *O*, é definido pelo conjunto *T = {parâmetros de entrada, instância O_i da ontologia}*, tal que, os *parâmetros de entrada* da ontologia representam uma consulta estruturada em lógica descritiva representada por uma CQ e, O_i representa a instância inicial da ontologia *O*, sendo consideradas tanto as instâncias de uma classe, quanto as suas subclasses e superclasses.

Conforme a Definição I, o conjunto de dados de teste *T* corresponde ao domínio de entrada de *O*, denotado por *D(O)*, que representa todas as possíveis consultas que podem ser utilizadas em *O*. Para a formalização de uma consulta para compor um dado de teste $t \in T$, utiliza-se a definição de Stojanovic et al. (2003) que define uma consulta para ontologias conforme a Definição Ia:

- **Definição Ia**: Uma *consulta* é escrita na forma:
  - *para todo X P*(*X*, *k*)
    com X sendo um vetor de variáveis (x_1, x_2, ..., x_n), *k* um vetor de constantes e *P* um vetor de relações (propriedades).
  - Por exemplo, para uma consulta *Q* "*para todo X trabalhaEm*(*x*, *EmpresaY*) e *temProfissão*(*x*, *Profissão*)" tem-se:
    $X = (x), k = (EmpresaY, Profissão), P = (P_1, P_2), P_1 = trabalhaEm, P_2 = temProfissão$.

A Definição I é utilizada como base para alcançar o objetivo do item (iv) (Seção 1.4) das contribuições desta tese, que se refere à definição de um método padronizado para a geração dos dados de teste (Seção 4.2) para a execução do teste de mutação para ontologias.

Tendo como base a definição de Questões de Competência apresentada na Subseção 2.1.4 e a formalização destas questões em lógica descritiva para compor o conjunto de dados de teste *T*, é possível considerar cada axioma lógico descritivo que representa um conceito do domínio abordado na ontologia como uma CQ. Considerando a ontologia OWL *travel* da Figura 2.6, o Exemplo 3.1 apresenta uma CQ em linguagem natural para consultar quais são as acomodações econômicas de acordo com esta ontologia.

Exemplo 3.1: CQ em linguagem natural para consultar acomodações econômicas na ontologia *travel* (autor, 2019)

```
Quais acomodações possuem avaliação com uma ou duas estrelas?
```

O Exemplo 3.2 apresenta a Questão de Competência do Exemplo 3.1 formulada em notação de lógica descritiva.

Exemplo 3.2: CQ em lógica descritiva para consultar acomodações econômicas na ontologia *travel* (autor, 2019)

```
Accommodation ⊓ (∃hasRating.(oneStarRating, twoStarRating))
```

A CQ do Exemplo 3.2 formulada em lógica descritiva corresponde exatamente ao axioma lógico descritivo apresentado no Exemplo 2.3, que descreve a classe *BudgetAccommodation* da

ontologia *travel*. Com base nesses exemplos é possível observar que os axiomas lógico-descritivos definidos na ontologia para representar os conceitos do domínio, correspondem às CQs definidas em lógica descritiva para realizar consultas na ontologia. Deste modo, tanto os axiomas lógicos originais *A* definidos na ontologia em teste, como os axiomas lógicos mutados *A'* podem ser considerados como CQs.

Para Vrandečić e Gangemi (2006), os resultados esperados não precisam necessariamente serem conhecidos no momento da elaboração dos dados de teste. Diante dessa afirmação e considerando a hipótese do programador competente do teste de mutação para programas (Delamaro et al., 2007), o resultado de uma ontologia, obtido após a aplicação de um dado de teste, compõe a formação de um caso de teste.

- **Definição II:** *Caso de teste* para uma ontologia O, é definido pelo conjunto $CT = \{T, O_o\}$, tal que O_o é obtido aplicando-se uma CQ à instância O_i. Assim, O_o representa o resultado da ontologia O, sendo consideradas tanto as instâncias de uma classe, quanto as suas subclasses e superclasses.

O resultado da ontologia, representado pelo resultado da aplicação de uma CQ, corresponde à definição do resultado de uma consulta conforme proposto por Stojanovic et al. (2003) e apresentada na Definição IIa:

- **Definição IIa**: Seja o conjunto Ω o conjunto de todas as instâncias que podem ser mostradas em uma ontologia, para uma consulta Q escrita na forma:
  - *para todo X* $P(X, k)$
    o resultado é um elemento do conjunto $R(Q) = X = \{(x_1, x_2, ..., x_n)\}$, de modo que $P(X, k)$ são demonstráveis, ou seja, cada instância da relação $r\{x_1, x_2, ..., x_n, k_1\}$, $r \in P$ existe no conjunto Ω.

- **Exemplo de dado de teste e caso de teste:** Tendo como base a ontologia *travel* da Figura 2.6, para um possível dado de teste pode-se considerar uma CQ para localizar nessa ontologia "*quais são as acomodações econômicas que possuam uma ou duas estrelas*". Essa CQ é apresentada no Exemplo 3.3, formulada em DL na sintaxe *OWL Manchester*.

Exemplo 3.3: Dado de teste representado em DL na sintaxe *OWL Manchester* (autor, 2019)

```
Accommodation and (hasRating some ({OneStarRating, TwoStarRating}))
```

Ao aplicar o dado de teste do Exemplo 3.3 na ontologia *travel*, é obtido o resultado de que as acomodações econômicas que possuem uma ou duas estrelas são "área de *camping*" e "sáfari", conforme mostrado no Exemplo 3.4.

Exemplo 3.4: Resultado da aplicação do dado de teste do Exemplo 3.3 na ontologia *travel* (autor, 2019)

```
- Superclasse                  - Subclasses
    - Accommodation                - Campground
                                   - Safari
```

De acordo com as Definições I e II, um caso de teste é composto pelo dado de teste apresentado no Exemplo 3.3 mais o resultado esperado apresentado no Exemplo 3.4.

Conforme Allemang e Hendler (2011), como as ontologias são baseadas na suposição de mundo aberto, podem haver novos conceitos a serem representados, supondo que sempre há mais informações que podem ser conhecidas em uma ontologia. Isto dificulta para o testador conhecer o resultado esperado antes da aplicação de um dado de teste na ontologia, sendo necessário de acordo com a Definição II, a análise do resultado obtido após a aplicação de um dado de teste para compor um caso de teste.

3.3 TESTE DE MUTAÇÃO DE ONTOLOGIAS OWL

O teste de mutação é um critério da técnica de teste de programas baseada em erros, onde modificações são introduzidas no código fonte de um programa através de operadores de mutação desenvolvidos especificamente para a linguagem de programação em que o software em teste foi escrito, para simular possíveis defeitos cometidos pelo desenvolvedor (DeMillo et al., 1978; Delamaro et al., 2007). Esta técnica de teste é aplicada para diversas linguagems de programação.

Para aplicar o teste de mutação de ontologias OWL, uma ontologia O a ser testada sofre pequenas alterações em sua estrutura de acordo com operadores de mutação preestabelecidos, dando origem a um conjunto de ontologias mutantes chamado O'. As alterações são realizadas nas estruturas física e lógica das ontologias por operadores de mutação, que alteram um conjunto A de axiomas que definem estruturas como hierarquia de classes, restrições, propriedades e indivíduos. Cada operador de mutação realiza uma mudança sintática simples com base nos erros típicos cometidos pelos programadores, ou com o objetivo de forçar determinados objetivos de teste. Em geral, os operadores de mutação estão associados a um tipo ou classe de erros que se pretende revelar em O. Cada operador de mutação gera mais de uma ontologia mutante, pois se O contém vários axiomas que estão no domínio do operador, então esse operador é aplicado a cada um desses axiomas, gerando uma ontologia mutante $o' \in O'$ com um axioma mutado A' oriundo de A para cada mutação realizada.

Após a geração dos mutantes é necessário definir um conjunto de dados de teste T (conforme Definição I, Seção 3.2) cuja qualidade se deseja avaliar. O conjunto T deve ser definido como consultas escritas em uma linguagem declarativa de consultas para ontologias ou baseadas em um formalismo de lógica descritiva. Cada dado de teste $t \in T$ é definido com o objetivo de que em pelo menos uma vez que seja aplicado em O, produza um resultado diferente da aplicação desse mesmo dado de teste em cada ontologia mutante $o' \in O'$, de modo a identificar o defeito simulado pelo operador de mutação.

Após a aplicação dos dados de teste e a comparação dos resultados entre O e todos os mutantes $o' \in O'$, é aplicado o escore de mutação para analisar o grau de adequação dos dados de teste utilizados. Dada a possibilidade da geração de ontologias mutantes inconsistentes, de acordo com a Definição IV, nesta tese o escore de mutação é adaptado do teste de mutação para programas considerando-se o número de ontologias mutantes geradas, mortas, definidas como equivalentes e inconsistentes. Assim, o teste de mutação de ontologias OWL consiste na execução das seguintes etapas: a) geração das ontologias mutantes; b) geração dos dados de teste; c) teste da ontologia original e das ontologias mutantes; e d) análise dos mutantes, conforme é apresentado na Figura 3.2 o processo de execução do teste de mutação para ontologias OWL.

3.3.1 Etapas do teste de mutação de ontologias OWL

A seguir é apresentada a definição para a execução de cada uma das etapas do teste de mutação de ontologias OWL apresentadas na Figura 3.2.

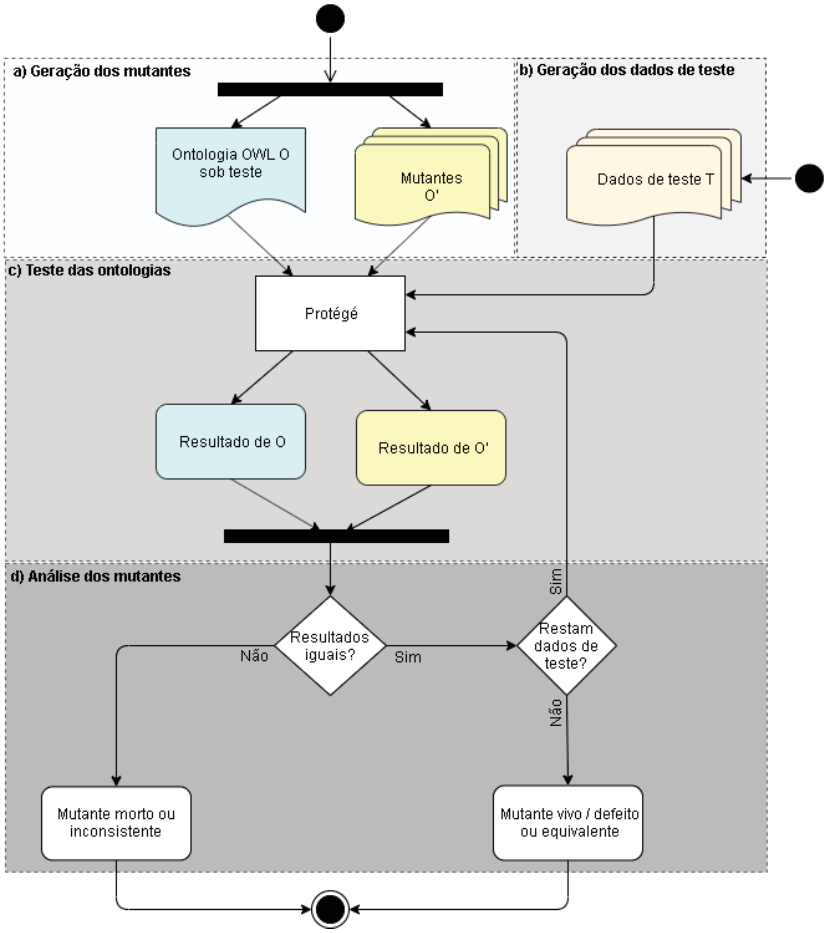


Figura 3.2: Teste de mutação de ontologias OWL com a ferramenta *Protégé* (autor, 2019)

a) Geração das ontologias mutantes: dada uma ontologia OWL *O* que tem por objetivo ser testada, os operadores de mutação são selecionados de acordo com os tipos de axiomas definidos nesta ontologia. Cada operador de mutação selecionado é aplicado em *O* produzindo uma mutação para cada axioma compatível com o operador. Para cada alteração realizada em *O* é gerada uma nova ontologia OWL mutante om_{ij}, dando origem ao conjunto de mutantes *O'*, conforme as definições III e IV apresentadas na sequência:

- **Definição III:** Aplicando-se um operador de mutação ao conjunto de axiomas *A* da ontologia *O*, obtém-se o *conjunto de mutantes* definido por $O' = \{om_{11}, om_{12}, ..., om_{i1}, om_{i2}, ... om_{ij}\}$, oriundo do conjunto *A*, tal que: om_{ij} é o j-ésimo mutante obtido a partir do operador de mutação $om_i \in OM$.
- **Definição IV:** A partir do conjunto *O'*, deriva-se o conjunto de mutantes consistentes *O'**, tal que $|O'^*| \leq |O'|$. Um mutante é consistente se for sintático e logicamente válido.

b) Geração de dados de teste: tem como objetivo encontrar um conjunto de dados de teste *T* capaz de matar os mutantes gerados pelos operadores de mutação. De acordo com a Definição I da Seção 3.2, o domínio de entrada de uma ontologia *O*, denotado por *D(O)*, é o conjunto de todos os valores possíveis representados por consultas formalizadas em lógica descritiva ou questões de competência que podem ser utilizados em *O*. É necessário encontrar subconjuntos reduzidos de *D(O)*, tal que $D(O) \in T$, para aplicar à

ontologia dependendo da cardinalidade de *D(O)*. Deste modo, conforme a definição de dado de teste apresentada na Seção 3.2, cada axioma $\{a \in A, A' \in A \mid A \in O\}$ dão origem aos subdomínios de *D(O)*. Como consequência, as mutações realizadas por cada operador de mutação determinam os subconjuntos de dados de teste com maiores possibilidades de identificar os defeitos simulados pelos operadores de mutação (Seção 3.4), dado que o operador de mutação produzirá os dados de teste de acordo com as mutações realizadas nos mutantes gerados por este operador.

De acordo com Delamaro et al. (2007), o dado de teste deve ser formado por aqueles dados capazes de distinguir o comportamento do programa original e do programa mutante. Assim, cada subconjunto de dados de teste gerado por cada operador de mutação está relacionado com a capacidade de revelar um defeito específico associado ao operador de mutação. Com esta abordagem evita-se executar a ontologia inteira para decidir quando um mutante é morto, de modo que cada subconjunto de dados de teste está relacionado ao critério de mutação fraca. Conforme Howden (1982), o critério de mutação fraca requer que um dado de teste $t \in T$ seja produzido de modo que, em pelo menos uma vez que seja aplicado em *A'*, produza um resultado diferente de *O*.

c) Teste das ontologias: cada dado de teste $t \in T$ é aplicado em *O* com o objetivo de analisar o resultado obtido. Caso seja retornado um resultado incorreto, então um defeito foi revelado em *O*. Cada *t* aplicado em *O* é também aplicado em cada mutante $o' \in O'$ com o objetivo de produzir um resultado diferente de *O* com o mesmo dado de teste *t*, de modo a identificar o defeito simulado pelo operador de mutação. Cada *t* é aplicado em *O* e em cada ontologia mutante *o'* com o auxílio de um classificador, o qual analisará o dado de teste de acordo com as restrições e axiomas definidos na ontologia, retornando um resultado com base na instanciação de classes e de indivíduos encontrados. Havendo um resultado diferente, *o'* é considerado morto e descartado, indicando que o defeito inserido pelo operador de mutação foi identificado pelo dado de teste. Caso *o'* seja um mutante inconsistente, ele também é descartado mas não é considerado como morto por um dado de teste *t*. Por outro lado, se *o'* apresenta resultado igual a *O*, considera-se *o'* como vivo, indicando uma fraqueza em *T*, por não revelar a diferença entre *o'* e *O*.

Nesta tese é usado o classificador *Pellet* (Sirin et al., 2007) em conjunto com a ferramenta *Protégé* (Musen, 2015), sendo ambos de código aberto, implementados na linguagem de programação Java e oferecem uma variedade de recursos, incluindo respostas de consultas conjuntivas, suporte de regras, raciocínio e apontamentos sobre axiomas.

Após o teste de todos os mutantes o escore de mutação é calculado para auxiliar o testador na decisão de parar o teste ou gerar novos dados de teste para continuar. Dada uma ontologia *O* e o conjunto de dados de teste *T*, calcula-se o escore de mutação com base no quociente entre o número de ontologias mutantes mortas e o resto entre o total de mutantes gerados e mutantes inconsistentes, conforme a Equação 3.1 a seguir:

$$ms(T) = \frac{Om}{(Og - Oi)} \tag{3.1}$$

- *Om* é o número de ontologias mutantes mortas;
- *Og* é o número de ontologias mutantes geradas;
- *Oi* é o número de ontologias mutantes inconsistentes.

De acordo com Delamaro et al. (2007), o escore de mutação varia entre 0 e 1 e fornece uma medida objetiva de quanto o conjunto de dados de teste analisado aproxima-se da adequação. Neste sentido, quanto mais próximo de 1 for o resultado do escore de mutação, mais confiáveis e adequados são os dados de teste utilizados, e o testador pode tomar a decisão de finalizar o teste.

d) Análise dos mutantes: para cada $o' \in O'$ definido como vivo, ou seja, que não apresentou um resultado que o diferencie de O com nenhum dado de teste $t \in T$, o' pode ter revelado um defeito existente em O, ou caso nenhum defeito seja identificado pode ser definido como equivalente a O. Analogamente ao teste de mutação para programas, definir se o' é equivalente a O é uma questão indecidível, assim, decidir se o teste continua enquanto o' não apresenta um resultado que o diferencie de O, fica a critério do testador.

De acordo com Delamaro et al. (2007), a tarefa de oráculo, ou seja, decidir se o resultado produzido está correto ou não, é realizada pelo testador. Esta abordagem coincide com a afirmação de Vrandečić e Gangemi (2006) de que os resultados esperados não precisam necessariamente serem conhecidos no momento da elaboração dos dados de teste. Assim, de acordo com a Defnição II da Seção 3.2, os casos de teste são obtidos após a execução de um dado de teste na ontologia original e a execução da tarefa de oráculo realizada pelo testador.

Após a definição das ontologias mutantes o' equivalentes à ontologia original O, o escore de mutação é calculado novamente para averiguar o grau de adequação e confiabilidade do conjunto de dados de teste em identificar os defeitos simulados pelos operadores de mutação. Assim, dada uma ontologia O e o conjunto de dados de teste T, calcula-se o escore de mutação com base no quociente entre o número de ontologias mutantes mortas e o resto entre o total de mutantes gerados, equivalentes a O e inconsistentes, conforme a Equação 3.2 a seguir:

$$ms(T) = \frac{Om}{(Og - Oe - Oi)} \tag{3.2}$$

- Om é o número de ontologias mutantes mortas;
- Og é o número de ontologias mutantes geradas;
- Oe é o número de ontologias mutantes equivalentes.
- Oi é o número de ontologias mutantes inconsistentes.

Um conjunto de dados de teste T é adequado à O em relação aos mutantes O', se para cada ontologia $o' \in O'$, ou $o' \equiv O$ ou $o' \neq O$ em pelo menos um dado de teste.

3.4 OPERADORES DE MUTAÇÃO PARA ONTOLOGIAS OWL

Como exposto por Delamaro et al. (2007), não existe uma maneira direta para definir os operadores de mutação, normalmente eles são projetados com base na experiência de uso da linguagem de programação à qual serão aplicados. De acordo com Derezinska (2003), operadores de mutação não precisam necessariamente serem aplicados a algum tipo de código implementado, desse modo, também são desenvolvidos para modelos como diagramas UML. Seguindo esse raciocínio e como a linguagem OWL adota um modelo da orientação a objetos (Horrocks, 2008), a seguir é apresentado como é definido neste trabalho um conjunto de operadores de mutação para ontologias OWL, conforme segue na Definição V:

- **Definição V:** *Conjunto de operadores de mutação* é definido pelo conjunto $OM = \{om_1, om_2, ..., om_x\}$, formado por operadores de mutação aplicáveis a um conjunto A de axiomas *TBox* ou *Abox* (Seção 2.1) da ontologia O, tal que $x > 0$.

Em Derezinska (2003) são apresentados operadores de mutação para diagramas de classes UML. Dentre eles, operadores que realizam mutações nas hierarquias e associações entre classes, também mencionados por Lee et al. (2008) como mutação de superclasses e subclasses no contexto de Serviços *Web* (do inglês, *Web Services*) semânticos OWL-S. Seguindo essas duas abordagens, em Porn (2014) definimos um conjunto de 25 operadores de mutação para ontologias OWL, que realizam modificações na estrutura de classes e nas definições de domínio e intervalo das propriedades de objetos e propriedades de tipos de dados.

Tendo a Definição V como base para a construção de operadores de mutação para ontologias OWL, foi desenvolvido um conjunto de 38 operadores de mutação, que foram definidos como operadores de mutação estrutural e operadores lógicos de mutação para ontologias OWL. Os operadores de mutação apresentados em Porn (2014) foram analisados e readequados neste conjunto de operadores no contexto desta tese. Na Subseção 3.4.1 são apresentados os operadores de mutação estrutural e na Subseção 3.4.2 são apresentados os operadores lógicos de mutação.

3.4.1 Operadores de mutação estrutural para ontologias OWL

Os operadores de mutação estrutural estão associados ao conjunto C, P e I da definição formal de ontologias apresentada na Seção 2.1, sendo responsáveis por realizar mutações no conjunto de conceitos que definem o corpo do conhecimento do domínio da ontologia. A Definição VI representa este conjunto de operadores:

- **Definição VI:** *Conjunto de operadores de mutação estrutural* é definido pelo subconjunto $OME = \{ome_1, ome_2, ..., ome_x\} \in OM$, formado por operadores de mutação aplicáveis a um conjunto Co de conceitos compostos pelo conjunto $\{C, P$ ou $I\}$ da definição formal de uma ontologia O (Seção 2.1), tal que $x > 0$.

A seguir são apresentados os operadores de mutação estrutural para ontologias OWL.

3.4.1.1 Operador de mutação ClassUpCascade - CUC

- **Objetivo:** Alterar a estrutura hierárquica de uma classe sem alterar as suas subclasses.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes SP_1, SP_2 e C, tal que SP_1, SP_2 e $C \in OWLClass$ e existe relação do tipo *subClassOf* entre SP_2 e SP_1 (SP_2 é subclasse de SP_1) e entre C e SP_2 (C é subclasse de SP_2), então a relação *subClassOf* entre C e SP_2 pode ser alterada para *subClassOf* entre C e SP_1 (C passa a ser subclasse de SP_1 e não mais de SP_2); para toda classe SB_n, tal que $SB_n \in OWLClass$ e existe relação *subClassOf* entre SB_n e C (SB_n é subclasse de C), então a relação *subClassOf* entre SB_n e C deve ser mantida.
- **Descrição:** Uma classe C, subclasse de SP_2, sobe um nível na hierarquia e passa a ficar associada imediatamente à superclasse de sua superclasse original (SP_1); suas subclasses SB_n acompanham esta mudança, ou seja, permanecem associadas à classe C.
- **Exemplo:** A classe *Profissional* é subclasse de *Pessoa* e tem ainda duas subclasses: *Enfermeiro* e *Médico*. *Profissional* sobe na hierarquia e passa a estar no mesmo nível de *Pessoa*, e suas subclasses *Enfermeiro* e *Médico* continuam sendo subclasses de *Profissional*.

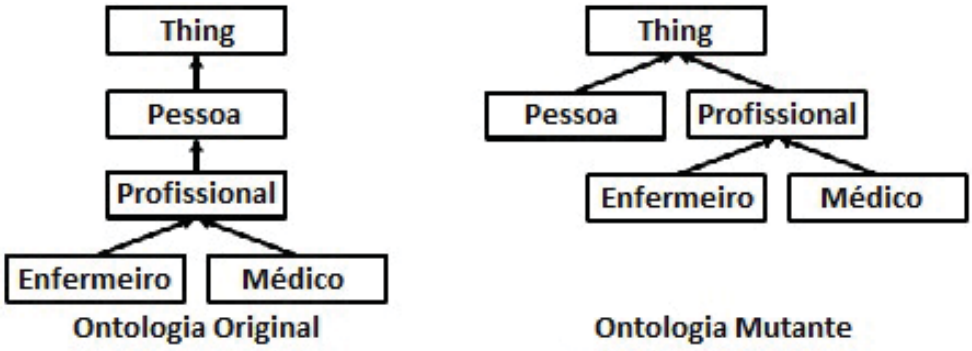


Figura 3.3: Modelo de mutação com o operador CUC (autor, 2019)

Os defeitos associados ao operador de mutação CUC são apresentados na Tabela 3.1.

Tabela 3.1: Defeitos associados ao operador de mutação CUC

Defeito	Motivo
Defeito circulatório (Gómez-Pérez, 2004)	Ao realizar a mutação na hierarquia das classes, a classe modificada pode tornar-se subclasse ou superclasse dela mesma, conforme definição de outros axiomas.
Defeito de partição (Gómez-Pérez, 2004)	Ao realizar a mutação na hierarquia das classes, é possível que a classe modificada torne-se subclasse de duas classes disjuntas, podendo invalidar a ontologia.
Definição de elementos recursivos (Poveda-Villalón et al., 2010)	Caso uma classe torne-se superclasse ou subclasse dela mesma, este defeito pode ser revelado devido a ocorrência da recursão apresentada no componente.
Definição formal idêntica de classes e instâncias (Gómez-Pérez, 2004)	Caso duas ou mais classes possuam a mesma representação conceitual, ao alterar a superclasse de uma dessas classes, poderá não ocorrer nenhuma alteração após a aplicação do *reasoner* .

3.4.1.2 Operador de mutação ClassDisjointDef - CDD

- **Objetivo:** Adicionar uma definição de disjunção entre um grupo de classes.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo de ontologias *OWL*, dada uma classe C, tal que C pertence a *OWLClass*, e um conjunto de classes SB_n, tal que $SB_n \in OWLClass$ e existe relação do tipo *subClassOf* entre cada uma das classes SB_n e C (SB_n é subclasse de C), e não existe relação *disjointWith* entre as classes SB_n, então é criada uma relação *disjointWith* entre todas as classes SB_n tornando-as disjuntas.
- **Descrição:** O operador CDD cria uma definição de disjunção entre classes, para todas as subclasses SB_n de uma classe C.
- **Exemplo:** As classes *Enfermeiro* e *Médico* são subclasses da classe *Profissional* e não estão definidas como classes disjuntas; essas classes passam a ser definidas disjuntas invalidando qualquer pessoa instanciada para as classes *Médico* e *Enfermeiro* simultaneamente.

Figura 3.4: Modelo de mutação com o operador CDD (autor, 2019)

Os defeitos associados ao operador de mutação CDD são apresentados na Tabela 3.2.

Tabela 3.2: Defeitos associados ao operador de mutação CDD

Defeito	Motivo
Classificação incompleta (Gómez-Pérez, 2004)	Pode-se inferir que houve a omissão de disjunção na definição das classes da ontologia original.
Falta de disjunção (Poveda-Villalón et al., 2010)	Infere-se que houve a omissão de disjunção na definição das classes da ontologia original.
Informação básica incompleta (Poveda-Villalón et al., 2010)	Infere-se que a disjunção é uma característica básica da classe mutada.

3.4.1.3 Operador de mutação ClassEquivalentUndef - CEU

- **Objetivo:** Remover a definição de equivalência de uma classe.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* entre as classes C e D, então essa relação *equivalentClass* é removida da definição da ontologia. A classe C passa a não ser mais descrita por uma relação *equivalentClass*.
- **Descrição:** Para uma classe C descrita por um axioma de equivalência, o operador CEU remove essa definição de equivalência, de modo que C passa a não ser mais descrita por esse axioma.
- **Exemplo:** A classe *Gestante* possui a definição de equivalência *"Pessoa and (éGestante value true)"*. Essa definição é removida fazendo com que as *Pessoas* que possuem a propriedade *éGestante* com valor *true* não sejam mais classificadas como pertencentes a classe *Gestante*.

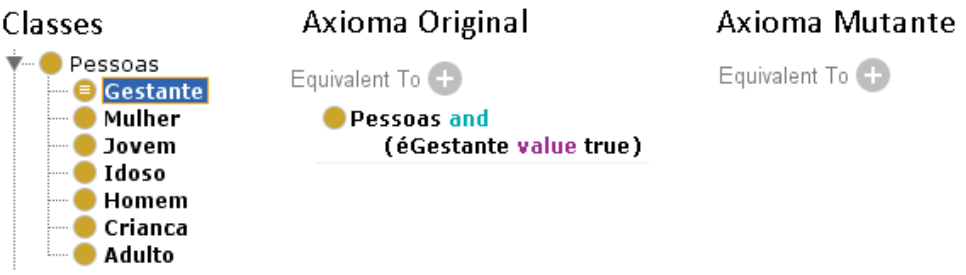


Figura 3.5: Modelo de mutação com o operador CEU (autor, 2019)

Os defeitos associados ao operador de mutação CEU são apresentados na Tabela 3.3.

Tabela 3.3: Defeitos associados ao operador de mutação CEU

Defeito	Motivo
Assumir característica implícita no nome dos componentes (Rector et al., 2004)	Pode ocorrer de apenas o nome da classe permanecer como sendo seu único identificador.
Uso incorreto de classes primitivas e definidas (Poveda-Villalón et al., 2010)	Para uma classe descrita com axiomas primitivos e definidos, ao remover a definição de equivalência, essa classe é representada somente como primitiva.
Redundância gramatical (Gómez-Pérez, 2004)	A classe mutada pode tornar-se similar a outra classe, principalmente quando o seu nome for o seu único identificador.
Criação de elementos polissêmicos (Poveda-Villalón et al., 2010)	Objetos obrigatórios não serão mais instanciados à classe ou objetos indevidos passam a fazer parte dela.
Criar uma classe para representar diferentes conceitos (Poveda-Villalón et al., 2010)	Diferentes objetos podem ser instanciados para a mesma classe, passando a representar diferentes conceitos.

3.4.1.4 Operador de mutação PropertyDomainDown - PDD

- **Objetivo:** Alterar a definição de domínio para uma classe um nível abaixo na hierarquia.
- **Definição:** Considerando o conjunto de propriedades *Property* do metamodelo *OWL* (que inclui tanto *OWLObjectProperty* quanto *OWLDatatypeProperty*), dada uma propriedade P e uma classe C, tal que $P \in Property$ e $C \in OWLClass$, existe relação do tipo *hasValueDomain* entre P e C, e existe relação do tipo *subClassOf* entre C e pelo menos outra classe C_2 do conjunto *OWLClass* (C possui subclasses), então a relação *hasValueDomain* entre P e C pode ser alterada para *hasValueDomain* entre P e C_2 (o domínio da propriedade P passa a ser restrito a C_2).
- **Descrição:** Este operador troca o domínio de uma propriedade por uma subclasse da classe originalmente definida como sendo este domínio, com o objetivo de testar se essa definição não foi feita de forma genérica para a propriedade em questão.
- **Exemplo:** A propriedade *éGestante* possui seu domínio definido com a classe *Pessoas*. O domínio é alterado para cada uma de suas subclasses, como *Adulto*, *Criança*, *Homem*, *Idoso*, *Jovem* e *Mulher*, o que faz com que apenas um destes subtipos possa ter a propriedade definida em seu escopo.

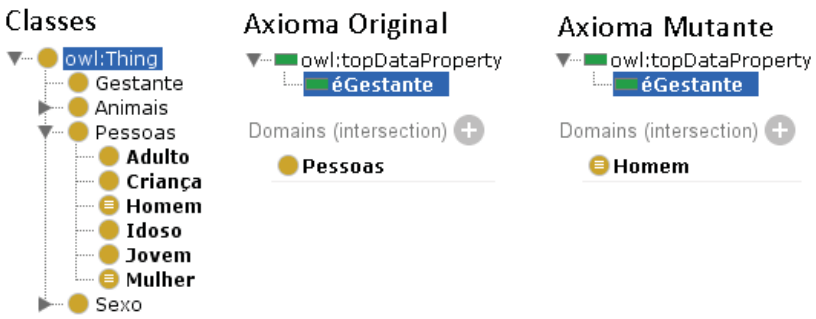


Figura 3.6: Modelo de mutação com o operador PDD (autor, 2019)

Os defeitos de modelagem de ontologias associados a este operador de mutação são: “Domínio e intervalo de propriedades” (Rector et al., 2004), devido a alteração do domínio de uma classe poder causar mudanças estruturais e de instanciação de objetos e, “Intervalo de dados ou domínio limitado” (Poveda-Villalón et al., 2010), pois ao alterar o domínio de uma classe é possível a obtenção de novos resultados na ontologia.

3.4.1.5 Operador de mutação PropertyDomainUp - PDUP

- **Objetivo:** Alterar o domínio das propriedades para uma classe acima na hierarquia.
- **Definição:** Considerando o conjunto de propriedades *Property* do metamodelo *OWL* (que inclui tanto *OWLObjectProperty* quanto *OWLDatatypeProperty*), dada uma propriedade P e uma classe C, tal que $P \in Property$ e $C \in OWLClass$, e existe relação do tipo *hasValueDomain* entre P e C, e existe relação do tipo *subClassOf* entre C e outra classe C_2 do conjunto *OWLClass* (C possui superclasse), então a relação *hasValueDomain* entre P e C pode ser alterada para *hasValueDomain* entre P e C_2 (o domínio da propriedade P passa a ser restrito a C_2).
- **Descrição:** Este operador troca o domínio de uma propriedade pela superclasse da classe originalmente definida como sendo este domínio, com o objetivo de testar se essa definição não foi feita de forma limitada para a propriedade em questão.
- **Exemplo:** A propriedade *éGestante* possui seu domínio definido com a classe *Pessoas*. O domínio é alterado para a classe *Thing* que é superclasse de *Pessoas*, fazendo com que as outras subclasses, como *Animais* ou *Sexo*, passem a fazer parte desta propriedade.

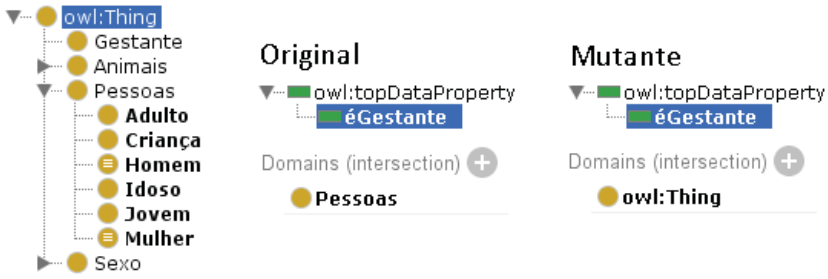


Figura 3.7: Modelo de mutação com o operador PDUP (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos apresentados para o operador de mutação PDD (Subseção 3.4.1.4), devido à similaridade das mutações realizadas.

3.4.1.6 Operador de mutação PropertyRangeDown - PRD

- **Objetivo:** Alterar o intervalo das propriedades para uma classe abaixo na hierarquia.
- **Definição:** Considerando o conjunto de propriedades *OWLObjectProperty* do metamodelo *OWL*, dada uma propriedade P e uma classe C, tal que $P \in OWLObjectProperty$ e $C \in OWLClass$, e existe relação do tipo *allValuesFrom* entre P e C, e existe relação do tipo *subClassOf* entre C e outra classe C_2 do conjunto *OWLClass* (C possui subclasses), então a relação *allValuesFrom* entre P e C pode ser alterada para *allValuesFrom* entre P e C_2 (o intervalo de valores possíveis da propriedade P passa a ser restrito a C_2).

- **Descrição:** Este operador troca o intervalo de uma propriedade por uma subclasse da classe originalmente definida como sendo este intervalo, com o objetivo de testar se essa definição não foi feita de forma genérica para a propriedade em questão.
- **Exemplo:** A propriedade *temSexo* possui intervalo definido com a classe *Sexo*. O intervalo é alterado para cada uma de suas subclasses, *Feminino* e *Masculino*, restringindo os indivíduos utilizados na associação desta propriedade com a classe de domínio.

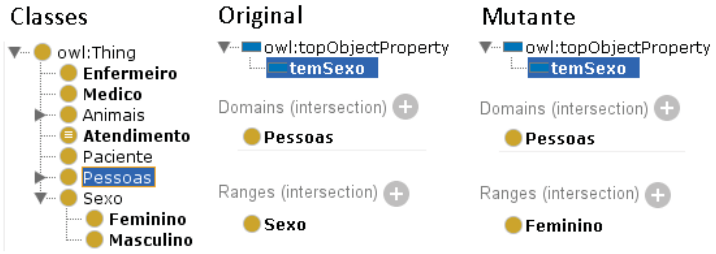


Figura 3.8: Modelo de mutação com o operador PRD (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos apresentados no operador de mutação PDD (Subseção 3.4.1.4), devido à similaridade das mutações realizadas.

3.4.1.7 Operador de mutação PropertyRangeUp - PRUP

- **Objetivo:** Alterar o intervalo das propriedades para uma classe acima na hierarquia.
- **Definição:** Considerando o conjunto de propriedades *OWLObjectProperty* do metamodelo *OWL*, dada uma propriedade P e uma classe C, tal que $P \in$ *OWLObjectProperty* e $C \in$ *OWLClass*, e existe relação do tipo *allValuesFrom* entre P e C, e existe relação do tipo *subClassOf* entre C e outra classe C_2 do conjunto *OWLClass* (C possui superclasse), então a relação *allValuesFrom* entre P e C pode ser alterada para *allValuesFrom* entre P e C_2 (C_2 é o intervalo de valores possíveis da propriedade P passa a ser restrito a C_2).
- **Descrição:** Este operador troca o intervalo de uma propriedade pela superclasse da classe originalmente definida como sendo este intervalo, com o objetivo de testar se essa definição não foi feita de forma limitada para a propriedade em questão.
- **Exemplo:** A propriedade *temSexo* possui seu intervalo definido com a classe *Sexo*. O intervalo é alterado para a sua superclasse *Thing*, deste modo todos os indivíduos das outras subclasses, como *Animais* ou *Sexo* podem ser utilizados na associação desta propriedade com a classe de domínio.

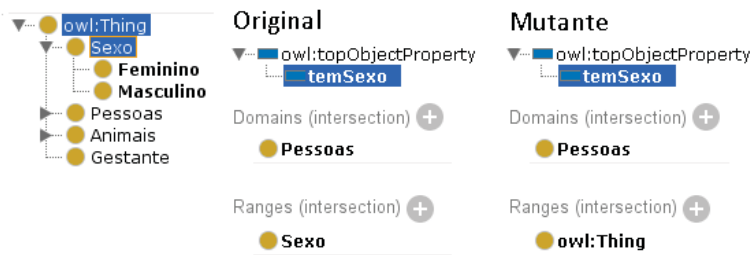


Figura 3.9: Modelo de mutação com o operador PRUP (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos apresentados no operador de mutação PDD (Subseção 3.4.1.4), devido à similaridade das mutações realizadas.

3.4.1.8 *Operador de mutação PropertyDomainUndef - PDU*

- **Objetivo:** Remover a definição de domínio nas propriedades de objetos e propriedades de tipos de dados.
- **Definição:** Considerando o conjunto de propriedades *Property* do metamodelo *OWL* (que inclui tanto *OWLObjectProperty* quanto *OWLDatatypeProperty*), dada uma propriedade P e uma classe C, tal que $P \in Property$ e $C \in OWLClass$, e existe relação do tipo *hasValueDomain* entre P e C, então a relação *hasValueDomain* entre P e C é removida, P passa a não ter nenhum tipo de relação *hasValueDomain*.
- **Descrição:** O operador de mutação PDU remove a definição de domínio de uma propriedade de objetos ou de uma propriedade de tipos de dados. Esta remoção do domínio da propriedade tem como objetivo testar se essa definição não foi feita de forma genérica para a propriedade em questão.
- **Exemplo:** A propriedade de tipos de dados *éGestante* possui seu domínio definido com a classe *Pessoas*, indicando que somente indivíduos dessa classe podem ser associados pela propriedade *éGestante*. Esse domínio é removido, possibilitando que as outras classes do modelo possam fazer parte de uma associação utilizando a propriedade de tipos de dados *temSexo*.

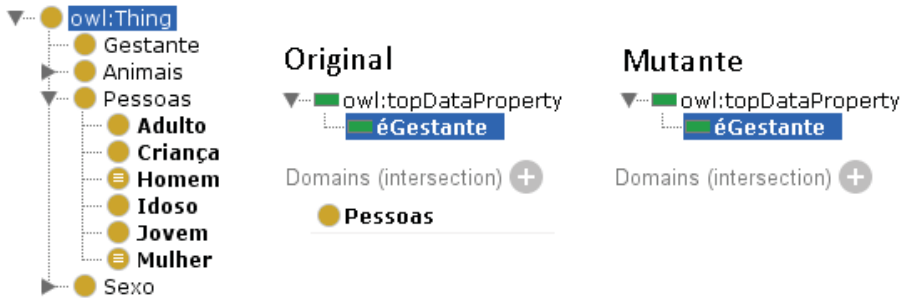


Figura 3.10: Modelo de mutação com o operador PDU (autor, 2019)

Os defeitos associados ao operador de mutação PDU são apresentados na Tabela 3.4.

Tabela 3.4: Defeitos associados ao operador de mutação PDU

Defeito	Motivo
Assumir característica implícita no nome do componente (Rector et al., 2004)	A propriedade passa a não ter mais relação direta com nenhum outro componente da ontologia.
Falta de domínio ou intervalo em propriedades (Poveda-Villalón et al., 2010)	A propriedade passa a não ter mais relação direta com nenhum outro componente da ontologia.
Criação de elementos desconexos (Poveda-Villalón et al., 2010)	A propriedade passa a não ter mais relação direta com nenhum outro componente da ontologia.
Intervalo de dados ou domínio limitado (Poveda-Villalón et al., 2010)	O domínio pode ter sido especificado de forma genérica.

3.4.1.9 *Operador de mutação PropertyRangeUndef - PRU*

- **Objetivo:** Remover a definição de intervalo nas propriedades de objetos.
- **Definição:** Considerando o conjunto de propriedades *OWLObjectProperty* do metamodelo *OWL*, dada uma propriedade P e uma classe C, tal que $P \in$ *OWLObjectProperty* e $C \in$ *OWLClass* e existe relação do tipo *allValuesFrom* entre P e C, a relação *allValuesFrom* entre P e C é removida, P passa a não ter nenhum tipo de relação *allValuesFrom*.
- **Descrição:** Este operador remove o intervalo de uma propriedade de objetos para testar se essa definição não foi feita de forma genérica para a propriedade em questão.
- **Exemplo:** A propriedade de objetos *temSexo* possui intervalo definido com a classe *Sexo*, indicando que cada indivíduo associado por esta propriedade tem como intervalo os indivíduos da classe *Sexo*. O intervalo é removido, deixando de restringir os indivíduos que poderiam ser utilizados na associação desta propriedade com a classe de domínio.

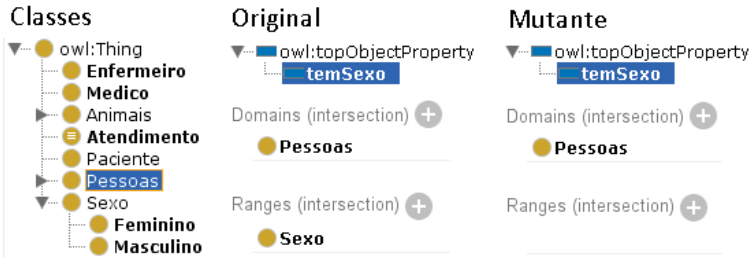


Figura 3.11: Modelo de mutação com o operador PRU (autor, 2019)

Os defeitos associados ao operador de mutação PRU são os mesmos apresentados na Tabela 3.4, devido a similaridade das mutações realizadas com o operador de mutação PDU.

Os operadores de mutação *ClassDisjointDefOne* - CDDO, *ClassEquivalentDef* - CED, *ClassDisjointUndefOne* - CDUO e *ClassDisjointUndefAll* - CDUA apresentados em Porn (2014), deram origem a novos operadores de mutação e foram descontinuados. O operador de mutação CDDO foi substituído pelos novos operadores de mutação estrutural *AxiomChangeSubClassToDisjoint* - ACSD e *AxiomChangeEquivalentToDisjoint* - ACED. O operador de mutação CED deu origem a outros dois novos operadores de mutação estrutural, sendo os operadores *PrimitiveToDefinedClass* - PDC e *DefinedToPrimiteClass* - DPC. Já os operadores de mutação CDUO e CDUA foram substituídos por um novo operador de mutação estrutural, dando origem ao operador *ClassDisjointUndef* - CDU. Esses 5 novos operadores de mutação estrutural seguem descritos abaixo.

3.4.1.10 *Operador de mutação AxiomChangeSubClassToDisjoint - ACSD*

- **Objetivo:** Alterar para *disjointWith* um axioma definido como *subClassOf*.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes SB_1 e C, tal que SB_1 e $C \in$ *OWLClass* e existe relação do tipo *subClassOf* entre SB_1 e C (SB_1 é subclasse de C), então a relação *subClassOf* entre C e SB_1 pode ser alterada para *disjointWith* entre C e SB_1 (SB_1 passa a ser uma classe disjunta de C). Para toda classe SB_n, tal que $SB_n \in$ *OWLClass* e existe relação *subClassOf* entre SB_n e C (SB_n é subclasse de C), então a relação *subClassOf* entre SB_n e C deve ser

alterara para *disjointWith* entre C e SB_n. A relação "*subClassOf*" passa a ser descrita na forma lógica apenas pelo operador "*disjointWith*".

- **Descrição:** Para uma classe descrita por um axioma lógico como "*C subClassOf D*", o operador ACSD altera esse axioma substituindo o operador OWL "*subClassOf*" pelo operador "*disjointWith*".
- **Exemplo:** A classe *Homem* possui o axioma lógico "*Homem subClassOf Pessoas and (temSexo some Masculino)*". Esse axioma é alterado para "*Homem disjointWith Pessoas and (temSexo some Masculino)*".

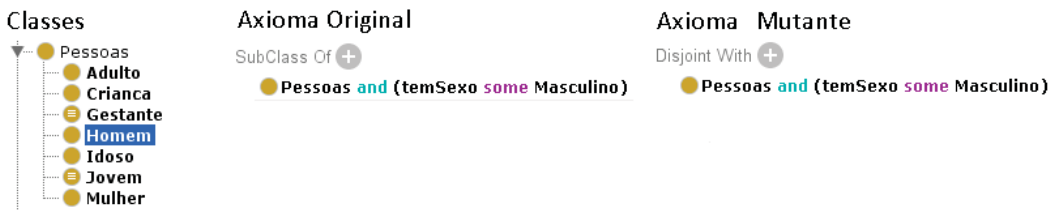


Figura 3.12: Modelo de mutação com o operador ACSD (autor, 2019)

Os defeitos associados ao operador de mutação ACSD são apresentados na Tabela 3.5.

Tabela 3.5: Defeitos associados ao operador de mutação ACSD

Defeito	Motivo
Inconsistênca semântica (Gómez-Pérez, 2004)	Ao definir um conceito com um axioma do tipo *subClassOf* esse conceito pode ser classificado como subclasse de outro conceito ao qual ele não pertence.
Definição formal idêntica de classes e instâncias (Gómez-Pérez, 2004)	Ao alterar um axioma do tipo *subClassOf* para *disjointWith* é possível identificar a semelhança entre diferentes conceitos caso a nova disjunção não cause impacto na ontologia.
Assumir característica implícita no nome do componente (Rector et al., 2004)	A nova definição de disjunção pode não causar impacto na ontologia por não existirem axiomas que descrevam o conceito abordado.
Utilizar elementos incorretamente (Poveda-Villalón et al., 2010)	A nova definição de disjunção pode revelar que uma classe foi definida equivocadamente como subclasse de outra.

3.4.1.11 Operador de mutação AxiomChangeEquivalentToDisjoint - ACED

- **Objetivo:** Alterar para *disjointWith* um axioma definido como *equivalentTo*.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação *equivalentTo* entre C e D, então essa relação *equivalentTo* é alterada para *disjointWith*. A relação *equivalentTo* passa a ser descrita na forma lógica apenas pelo operador *disjointWith*.
- **Descrição:** Para uma classe descrita por um axioma lógico como "*C equivalentTo D*", o operador ACED altera esse axioma substituindo o operador "*equivalentTo*" pelo operador "*disjointWith*".

- **Exemplo:** A classe *Homem* possui o axioma "*Homem equivalentTo Pessoas and (temSexo some Masculino)*". Esse axioma é alterado para "*Homem disjointWith Pessoas and (temSexo some Masculino)*".

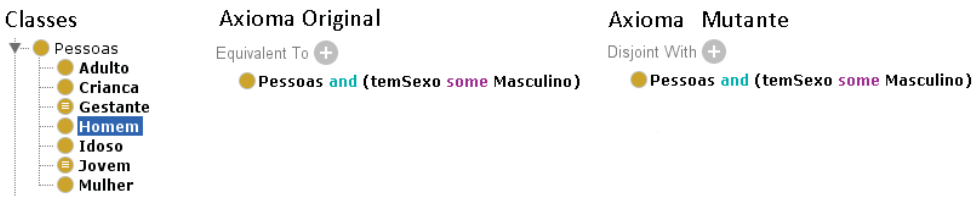


Figura 3.13: Modelo de mutação com o operador ACED (autor, 2019)

Os defeitos associados ao operador de mutação ACED são os mesmos apresentados para o operador ACSD (Subseção 3.4.1.10), devido à similaridade das mutações.

3.4.1.12 Operador de mutação PrimitiveToDefinedClass - PDC

- **Objetivo:** Converter uma classe primitiva para uma classe definida.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *subClassOf* entre C e D (C é subclasse de D), então essa relação *subClassOf* é convertida para uma relação "*equivalentTo*". A relação primitiva "*subClassOf*" passa a ser descrita na forma definida "*equivalentTo*".
- **Descrição:** Para uma classe primitiva descrita por um ou mais axiomas lógicos como "*C subClassOf D*", este operador converte esse axioma em um axioma definido *equivalentTo*.
- **Exemplo:** A classe *Homem* possui um axioma primitivo "*Pessoa and (temSexo some Masculino)*". Esse axioma é convertido para um axioma definido "*equivalentTo*".

Figura 3.14: Modelo de mutação com o operador PDC (autor, 2019)

Os defeitos associados ao operador de mutação PDC são os mesmos apresentados para o operador ACSD (Subseção 3.4.1.10), devido à similaridade das mutações.

3.4.1.13 Operador de mutação DefinedToPrimitiveClass - DPC

- **Objetivo:** Converter uma classe definida para uma classe primitiva.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentTo* entre C e D (C é equivalente a D), então essa relação *equivalentTo* é convertida para uma relação "*subClassOf*". A relação definida "*equivalentTo*" passa a ser descrita na forma primitiva "*subClassOf*".

- **Descrição:** Para uma classe definida descrita por um ou mais axiomas lógicos como "*C equivalentTo D*", este operador altera esse axioma em um axioma primitivo *subClassOf*.
- **Exemplo:** A classe *Homem* possui um axioma definido "*Pessoa and (temSexo some Masculino)*". Esse axioma é convertido para um axioma primitivo "*subClassOf*".

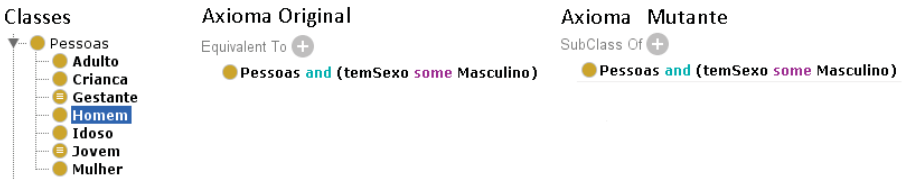


Figura 3.15: Modelo de mutação com o operador DPC (autor, 2019)

Os defeitos associados ao operador de mutação DPC são os mesmos apresentados para o operador ACSD (Subseção 3.4.1.10), devido à similaridade das mutações realizadas.

3.4.1.14 Operador de mutação ClassDisjointUndef - CDU

- **Objetivo:** Remover uma definição de disjunção entre um grupo de subclasses.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada a classe C e o conjunto de classes SB_n, tal que C e $SB_n \in OWLClass$ e existe relação *subClassOf* entre as classes SB_n e C (SB_n é subclasse de C), e existe relação *disjointWith* entre todas as classes SB_n, essa relação *disjointWith* de SB_n é removida.
- **Descrição:** O operador CDU remove uma definição de disjunção entre um conjunto de subclasses pertencentes a mesma superclasse.
- **Exemplo:** As classes *Homem* e *Mulher* estão definidas como disjuntas entre si, indicando que um indivíduo da classe *Homem* não pode fazer parte da classe *Mulher*. Essa definição de disjunção é removida. Essa remoção de disjunção além de permitir que um indivíduo do tipo *Homem* possa ser classificado como do tipo *Mulher*, a ontologia não reconhece mais os axiomas do tipo (*not Homem*) ou (*not Mulher*).

Figura 3.16: Modelo de mutação com o operador CDU (autor, 2019)

O defeito de modelagem de ontologias associado a este operador de mutação é o defeito de "Assumir características implícitas no nome dos componentes" (Rector et al., 2004), dado que o nome dos componentes da ontologia não são uma restrição suficiente para o classificador interpretar o significado do conceito pretendido.

Outros 8 operadores de mutação apresentados em Porn (2014) foram selecionados e classificados como operadores lógicos de mutação, sendo os operadores *ClassEquivalentUndefAnd* -

CEUA, *ClassEquivalentUndefOr* - CEUO, *AxiomChangeAndToOr* - ACATO, *AxiomChange-OrToAnd* - ACOTA, *AxiomChangeSomeToAll* - ACSTA, *AxiomChangeAllToSome* - ACATS, *AxiomEquivalentDefNot* - AEDN e *AxiomEquivalentUndefNot* - AEUN. Esses 8 operadores de mutação são apresentados na Subseção 3.4.2.

Dos 25 operadores de mutação apresentados em Porn (2014), 4 foram descontinuados sem dar origem a um novo operador. O operador de mutação *ClassUp* - CUP foi descontinuado por alterar de uma única vez a superclasse de uma classe e de suas subclasses, realizando um tipo de mutação de ordem k, sendo que o operador de mutação *ClassUpCascade* - CUC faz essa mesma mutação separadamente para cada conjunto de subclasses. O operador de mutação *ClassEquivalentCopyOne* - CECO mesmo estando em conformidade com a Definição V, ao invés de realizar uma mutação em um axioma da ontologia, ele cria uma nova estrutura idêntica a outra existente, gerando um novo conceito no domínio. Já os operadores *PropertyInverseDef* - PID e *PropertyInverseUndef* - PIUD não foram utilizados por não estarem em conformidade com as Definições VI e VII.

3.4.2 Operadores lógicos de mutação para ontologias OWL

Os operadores lógicos de mutação estão associados ao conjunto V e A da definição formal de ontologias, sendo responsáveis por realizar mutações no conjunto de axiomas definidos pela tripla (C, P, I) de acordo com a definição de ontologias dada na Seção 2.1. A Definição VII representa este conjunto de operadores:

- **Definição VII:** *Conjunto de operadores lógicos de mutação* é definido pelo subconjunto $OML = \{oml_1, oml_2, ..., oml_x\} \in OM$, formado por operadores de mutação aplicáveis a um conjunto A de axiomas compostos pela tripla $\{C, P, I\}$ da definição formal de uma ontologia O, tal que $x > 0$.

Na sequência são apresentados os operadores lógicos de mutação para onrologias OWL.

3.4.2.1 Operador de mutação ClassEquivalentUndefAnd - CEUA

- **Objetivo:** Remover o operando "AND" em uma definição de equivalência.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D (C é equivalente a D ou C é subclasse de D), relação esta descrita na forma lógica (C *and* D), então essa relação *equivalentClass* ou *subClassOf* pode ser alterada removendo-se um dos termos ligados pelo operando *"and"*. A relação *equivalentClass* ou *subClassOf* passa então a ser descrita na forma lógica apenas pelas expressões (C) ou (D).
- **Descrição:** Ao descrever uma classe como equivalente ou subclasse por um axioma lógico descritivo na forma (C *and* D), esse axioma lógico é alterado removendo-se o operando *AND*, mantendo-se somente como (C) ou (D).
- **Exemplo:** A classe *Gestante* é descrita por um axioma lógico de equivalência *"Pessoas and (éGestante value true)"*. Esse axioma é alterado para duas possíveis situações:
  - *"Pessoa"*: qualquer *pessoa* passaria a ser classificada como *Gestante*, inclusive todos os indivíduos da classe *Homem*, o que poderia incorrer em inconsistência em relação à definição de disjunção entre as classes *Homem* e *Mulher*.

– *"(éGestante value true)"*: todos os indivíduos instanciados com a propriedade *éGestante* definida com o valor *true* passariam a ser classificados como *Gestante*, independente de esses indivíduos serem instâncias da classe *Pessoas* ou não.

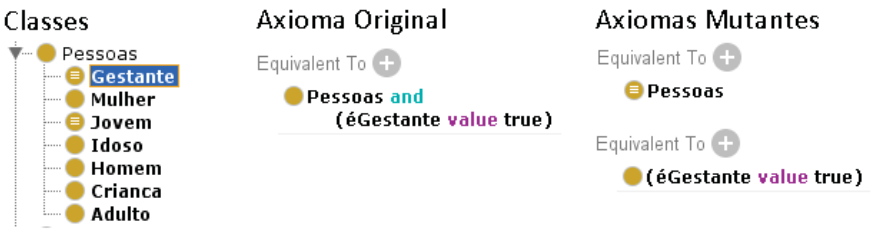


Figura 3.17: Modelo de mutação com o operador CEUA (autor, 2019)

Os defeitos associados ao operador de mutação CEUA são apresentados na Tabela 3.6.

Tabela 3.6: Defeitos associados ao operador de mutação CEUA

Defeito	Motivo
Uso incorreto de classes primitivas e definidas (Poveda-Villalón et al., 2010)	A classe mutada pode continuar sendo instanciada similarmente ao modelo antes da mutação.
Utilização incorreta dos operandos AND e OR (Rector et al., 2004)	A classe mutada pode continuar apresentando resultados similares ao modelo antes da mutação.
Redundância gramatical (Gómez-Pérez, 2004)	A nova definição torna-se similar a outra da ontologia.
Criação de elementos polissêmicos (Poveda-Villalón et al., 2010)	Pode ocorrer de a classe mutada não conseguir instanciar objetos que são obrigatórios a ela, ou pelo contrário, passa a instanciar objetos que não deveriam fazer parte dela.
Criação de classes sinônimas (Poveda-Villalón et al., 2010)	A classe pode tornar-se similar a outra já definida na ontologia, de modo que ambas sejam equivalentes.
Criar uma classe para representar diferentes conceitos (Poveda-Villalón et al., 2010)	Diferentes tipos de objetos podem ser instanciados para essa classe, passando assim a representar diferentes conceitos.

3.4.2.2 *Operador de mutação ClassEquivalentUndefOr - CEUO*

- **Objetivo:** Remover o operando "OR" em uma definição de equivalência.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D (C é equivalente a D ou C é subclasse de D), relação esta descrita de forma lógica (C *or* D), então essa relação pode ser alterada removendo-se um dos termos ligados pelo operando *"or"*. A relação passa então a ser descrita na forma lógica apenas pelas expressões (C) ou (D).
- **Descrição:** Ao definir uma classe como equivalente ou como subclasse por um axioma lógico na forma (C *or* D), o operador CEUO altera esse axioma mantendo a expressão somente como (C) ou (D).

- **Exemplo:** A classe *Jovem* possui definição de equivalência "*(temCaracteristica value Menina) or (temCaracteristica value Menino)*". Essa definição é alterada para "*(temCaracteristica value Menina)*" e "*(temCaracteristica value Menino)*".

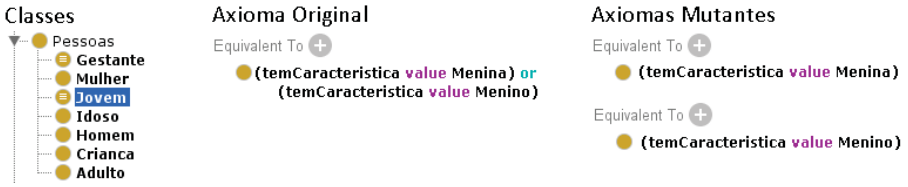


Figura 3.18: Modelo de mutação com o operador CEUO (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados na Tabela 3.6, devido as mutações serem similares as realizadas pelo operador de mutação CEUA.

3.4.2.3 Operador de mutação AxiomChangeAndToOr - ACATO

- **Objetivo:** Substituir o operando "AND" pelo operando "OR" em um axioma.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D (C é equivalente a D ou C é subclasse de D), relação esta descrita na forma lógica (C *and* D), então essa relação pode ser alterada para (C *or* D). A relação passa a ser descrita na forma lógica apenas pelo operando "*or*".
- **Descrição:** Ao definir uma classe como equivalente ou como subclasse por um axioma lógico na forma (C *and* D), o operador ACATO altera esse axioma lógico que define a equivalência da classe, substituindo o operando "*and*" pelo operando "*or*".
- **Exemplo:** A classe *Jovem* possui a equivalência *"Jovem and (temCaracteristica value Menino)"*. Essa definição é alterada para *"Jovem or (temCaracteristica value Menino)"*.

Figura 3.19: Modelo de mutação com o operador ACATO (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos apresentados na Tabela 3.6, devido as mutações serem similares as realizadas pelo operador de mutação CEUA (Subseção 3.4.2.1), diferenciando apenas que ao invés de remover o operando *AND* de um axioma, este operador de mutação substitui o operando *AND* pelo operando *OR* em uma definição.

3.4.2.4 Operador de mutação AxiomChangeOrToAnd - ACOTA

- **Objetivo:** Substituir o operando "OR" pelo operando "AND" em um axioma.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D (C é equivalente a D ou C é subclasse de D),

relação esta descrita na forma lógica (*C or D*), então essa relação pode ser alterada para (*C and D*). A relação passa a ser descrita na forma lógica apenas pelo operando "*and*".

- **Descrição:** Ao definir uma classe como equivalente ou como subclasse por um axioma lógico na forma (*C or D*), o operador ACOTA altera esse axioma lógico que define a equivalência da classe, substituindo o operando "*or*" pelo operando "*and*".

- **Exemplo:** A classe *Jovem* possui a equivalência "*Jovem or (temCaracteristica value Menino)*". Essa definição é alterada para "*Jovem and (temCaracteristica value Menino)*".

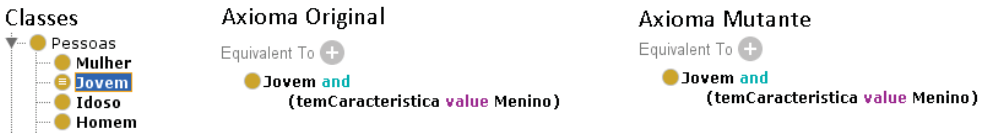


Figura 3.20: Modelo de mutação com o operador ACOTA (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos apresentados na Tabela 3.6, devido as mutações serem similares as realizadas pelo operador de mutação CEUA (Subseção 3.4.2.1), diferenciando apenas que ao invés de remover o operando *AND* de um axioma, este operador de mutação substitui o operando *OR* pelo operando *AND* em uma definição.

3.4.2.5 Operador de mutação AxiomChangeAllToSome - ACATS

- **Objetivo:** Substituir o operando *allValuesFrom* por *someValuesFrom* em um axioma.

- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes *C* e *D*, tal que *C* e *D* ∈ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre *C* e *D* (*C* é equivalente a *D* ou *C* é subclasse de *D*), relação esta descrita na forma lógica (*C allValuesFrom D*), então essa relação pode ser alterada para (*C someValuesFrom D*). A relação *equivalentClass* ou *subClassOf* passa a ser descrita na forma lógica apenas pelo operando "*someValuesFrom*".

- **Descrição:** Ao definir uma classe como equivalente ou subclasse por um axioma na forma (*C allValuesFrom D*), o operador ACATS altera esse axioma substituindo o operando "*allValuesFrom*" pelo operando "*someValuesFrom*".

- **Exemplo:** A classe *Atendimento* possui definição de equivalência "*(temPessoaEnvolvida allValuesFrom Profissional)*". Essa definição é alterada para "*(temPessoaEnvolvida someValuesFrom Profissional)*".

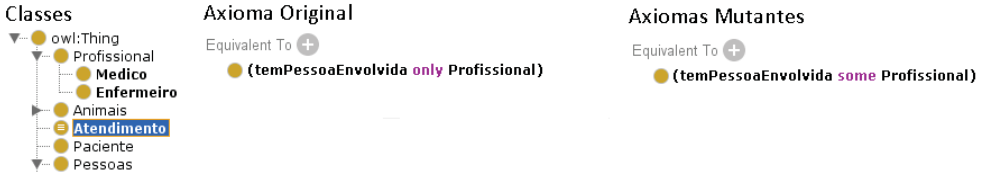


Figura 3.21: Modelo de mutação com o operador ACATS (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos apresentados na Tabela 3.6, exceto para o defeito de "Utilização incorreta dos operandos *AND* e *OR*" (Rector

et al., 2004). As mutações aplicadas são similares as realizadas pelo operador de mutação CEUA (Subseção 3.4.2.1), diferenciando apenas que ao invés de remover o operando *AND* de um axioma, este operador de mutação substitui o qualificador "*allValuesFrom*" pelo qualificador "*someValuesFrom*" em uma definição. Outro defeito que pode ser revelado é o "Uso equivocado de qualificadores de restrição", caso os resultados obtidos após a mutação sejam iguais aos do modelo original, devido o desenvolvedor ter utilizado o qualificador erroneamente.

3.4.2.6 Operador de mutação AxiomChangeSomeToAll - ACSTA

- **Objetivo:** Substituir o operando *someValuesFrom* por *allValuesFrom* em um axioma.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D (C é equivalente a D ou C é subclasse de D), relação esta descrita na forma lógica (C *someValuesFrom* D), então essa relação pode ser alterada para (C *allValuesFrom* D). A relação *equivalentClass* ou *subClassOf* passa a ser descrita na forma lógica apenas pelo operando "*allValuesFrom*".
- **Descrição:** Ao definir uma classe como equivalente ou subclasse por um axioma na forma (C *someValuesFrom* D), o operador ACSTA altera esse axioma substituindo o operando "*someValuesFrom*" pelo operando "*allValuesFrom*".
- **Exemplo:** A classe *Atendimento* possui definição de equivalência "*(temPessoaEnvolvida someValuesFrom Profissional)*". Essa definição é alterada para "*(temPessoaEnvolvida allValuesFrom Profissional)*".

Figura 3.22: Modelo de mutação com o operador ACSTA (autor, 2019)

Os possíveis defeitos revelados por este operador de mutação são os mesmos apresentados na Tabela 3.6, exceto para o defeito de "Utilização incorreta dos operandos *AND* e *OR*" (Rector et al., 2004). As mutações aplicadas são similares as realizadas pelo operador de mutação CEUA (Subseção 3.4.2.1), diferenciando apenas que ao invés de remover o operando *AND* de um axioma, este operador de mutação substitui o qualificador "*someValuesFrom*" pelo qualificador "*allValuesFrom*" em uma definição. Outro defeito que pode ser revelado é o "Uso equivocado de qualificadores de restrição", caso os resultados obtidos após a mutação sejam iguais aos do modelo original, devido o desenvolvedor ter utilizado o qualificador erroneamente.

3.4.2.7 Operador de mutação AxiomEquivalentDefNot - AEDN

- **Objetivo:** Adicionar um operando de negação na definição de equivalência entre classes.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo

equivalentClass ou *subClassOf* entre *C* e *D* descrita na forma lógica (*C and D*), (*C or D*), (*C someValuesFrom D*) ou (*C allValuesFrom D*), então essa relação é alterada para (*C and (not D)*), (*C or (not D)*), (*C someValuesFrom (not D)*) ou (*C allValuesFrom (not D)*). A relação *equivalentClass* ou *subClassOf* passa a ser descrita na forma lógica com o operando "*not*".

- **Descrição:** Ao definir uma classe como equivalente ou como subclasse por um axioma lógico na forma (*C allValuesFrom D*), (*C someValuesFrom D*), (*C and D*) ou (*C or D*), o operador AEDN altera esse axioma lógico que define a classe, adicionando o operando "*not*".

- **Exemplo:** A classe *Atendimento* tem equivalência "*(temPessoaEnvolvida allValuesFrom Profissional)*". Altera-se para "*(temPessoaEnvolvida allValuesFrom (not Profissional))*".

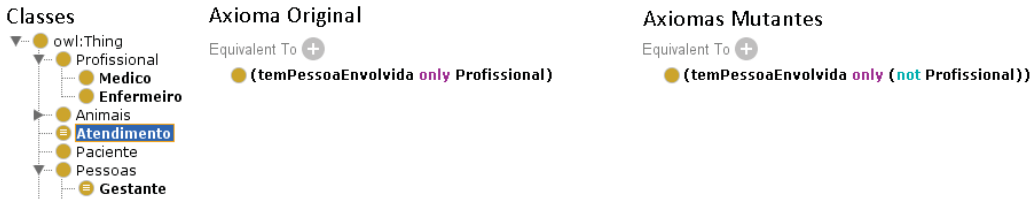


Figura 3.23: Modelo de mutação com o operador AEDN (autor, 2019)

Os defeitos associados a este operador são o "Uso incorreto da definição *some not*" (Poveda-Villalón et al., 2010), pois ao aplicar a negação em uma equivalência ou subclasse, este defeito pode ser revelado pela ausência de negação no axioma original ou pelo uso incorreto de uma disjunção. O defeito de "Utilizar elementos incorretamente" (Poveda-Villalón et al., 2010), caso uma negação tenha sido representada por uma disjunção ou até mesmo a ausência desta e, o defeito de "Assumir características implícitas no nome dos componentes" (Rector et al., 2004), caso ao definir a negação na representação de um conceito, não cause impacto na ontologia devido a outros conceitos estarem sendo representados somente pelo seu nome.

3.4.2.8 Operador de mutação AxiomEquivalentUndefNot - AEUN

- **Objetivo:** Remover um operando de negação na definição de equivalência entre classes.

- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes *C* e *D*, tal que *C* e *D* $\in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre *C* e *D*, relação esta descrita na forma lógica (*C and (not D)*), (*C or (not D)*), (*C someValuesFrom (not D)*) ou (*C allValuesFrom (not D)*), então essa relação é alterada para (*C and D*), (*C or D*), (*C someValuesFrom D*) ou (*C allValuesFrom D*). A relação *equivalentClass* ou *subClassOf* passa a ser descrita na forma lógica sem o operando "*not*".

- **Descrição:** Ao definir uma classe como equivalente ou como subclasse por um axioma lógico na forma (*C allValuesFrom (not D)*), (*C someValuesFrom (not D)*), (*C and (not D)*) ou (*C or (not D)*), o operador AEUN altera esse axioma lógico que define a classe removendo o operando "*not*".

- **Exemplo:** A classe *Atendimento* possui definição de equivalência "*(temPessoaEnvolvida allValuesFrom (not Profissional))*". Essa definição é alterada para "*(temPessoaEnvolvida allValuesFrom Profissional)*".

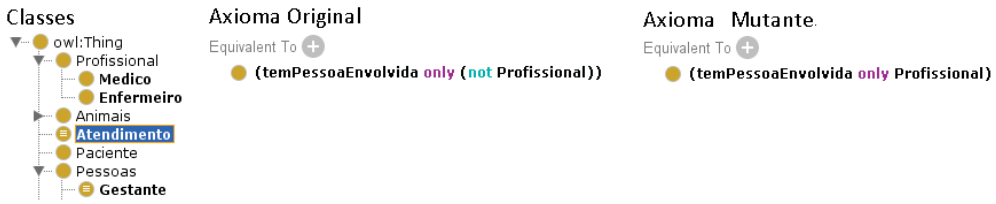


Figura 3.24: Modelo de mutação com o operador AEUN (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos apresentados para o operador AEDN (Subseção 3.4.2.7), devido à similaridade das mutações realizadas.

3.4.2.9 Operador de mutação AxiomChangeMinToMax - ACMiMa

- **Objetivo:** Substituir o operando "minCardinality" pelo operando "maxCardinality".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *minCardinality* D), então essa relação é alterada para "C *maxCardinality* D". A relação passa a ser descrita na forma lógica apenas pelo operando "*maxCardinality*".
- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "C *minCardinality* D", o operador ACMiMa altera esse axioma lógico substituindo o operando "*minCardinality*" pelo operando "*maxCardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*(temPessoaEnvolvida minCardinality 1 Paciente)*". Altera-se para "*(temPessoaEnvolvida maxCardinality 1 Paciente)*".

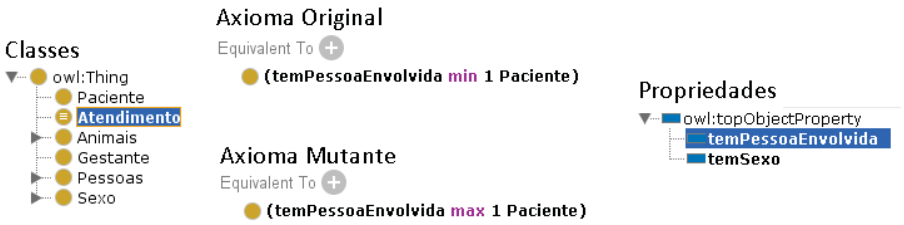


Figura 3.25: Modelo de mutação com o operador ACMiMa (autor, 2019)

O defeito associado a este operador de mutação é o defeito de "definição formal idêntica de classes e instâncias" (Gómez-Pérez, 2004), dada a possibilidade da ocorrência de classes ou indivíduos serem descritos de forma idêntica com operandos de cardinalidade.

3.4.2.10 Operador de mutação AxiomChangeMaxToMin - ACMaMi

- **Objetivo:** Substituir o operando "maxCardinality" pelo operando "minCardinality".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *maxCardinality* D), então essa relação é alterada para "C *minCardinality* D". A relação passa a ser descrita na forma lógica apenas pelo operando "*minCardinality*".

- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "*C maxCardinality D*", o operador ACMaMi altera esse axioma lógico substituindo o operando "*maxCardinality*" pelo operando "*minCardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*(temPessoaEnvolvida maxCardinality 1 Paciente)*". Altera-se para "*(temPessoaEnvolvida minCardinality 1 Paciente)*".

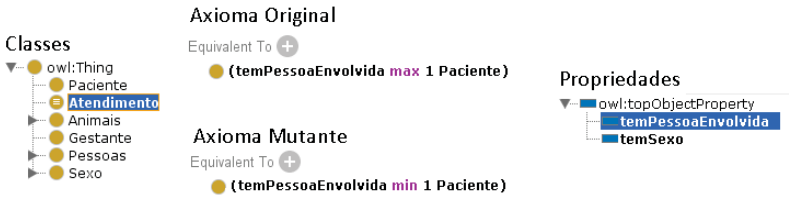


Figura 3.26: Modelo de mutação com o operador ACMaMi (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação ACMiMa (Subseção 3.4.2.9), devido à similaridade das mutações.

3.4.2.11 Operador de mutação AxiomChangeMinToExactly - ACMiEx

- **Objetivo:** Substituir o operando "minCardinality" pelo operando "Cardinality".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (*C minCardinality D*), então essa relação é alterada para "*C Cardinality D*". A relação passa a ser descrita na forma lógica apenas pelo operando "*Cardinality*".
- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "*C minCardinality D*", o operando "*minCardinality*" é substituído por "*Cardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*(temPessoaEnvolvida minCardinality 1 Paciente)*". Altera-se para "*(temPessoaEnvolvida Cardinality 1 Paciente)*".

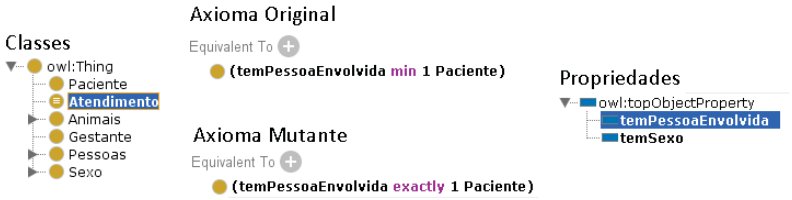


Figura 3.27: Modelo de mutação com o operador ACMiEx (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação ACMiMa (Subseção 3.4.2.9), devido à similaridade das mutações.

3.4.2.12 Operador de mutação AxiomChangeMaxToExactly - ACMaEx

- **Objetivo:** Substituir o operando "maxCardinality" pelo operando "Cardinality".

- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes *C* e *D*, tal que *C* e *D* ∈ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre *C* e *D*, relação esta descrita na forma lógica (*C maxCardinality D*), então essa relação é alterada para "*C Cardinality D*". A relação passa a ser descrita na forma lógica apenas pelo operando "*Cardinality*".
- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "*C maxCardinality D*", o operador ACMaEx altera esse axioma lógico substituindo o operando "*maxCardinality*" pelo operando "*Cardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*(temPessoaEnvolvida maxCardinality 1 Paciente)*". Altera-se para "*(temPessoaEnvolvida Cardinality 1 Paciente)*".

Figura 3.28: Modelo de mutação com o operador ACMaEx (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação ACMiMa (Subseção 3.4.2.9), devido à similaridade das mutações.

3.4.2.13 Operador de mutação AxiomChangeExactlyToMin - ACEMi

- **Objetivo:** Substituir o operando "Cardinality" pelo operando "minCardinality".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes *C* e *D*, tal que *C* e *D* ∈ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre *C* e *D*, relação esta descrita na forma lógica (*C Cardinality D*), então essa relação é alterada para "*C minCardinality D*". A relação passa a ser descrita na forma lógica apenas pelo operando "*minCardinality*".
- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "*C Cardinality D*", o operador ACEMi altera esse axioma lógico, substituindo o operando "*Cardinality*" pelo operando "*minCardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*(temPessoaEnvolvida Cardinality 1 Paciente)*". Altera-se para "*(temPessoaEnvolvida minCardinality 1 Paciente)*".

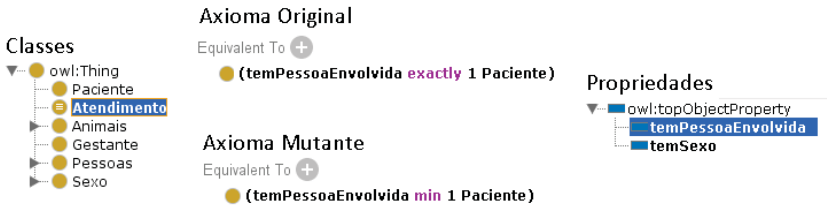


Figura 3.29: Modelo de mutação com o operador ACEMi (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação ACEMiMa (Subseção 3.4.2.9), devido à similaridade das mutações.

3.4.2.14 *Operador de mutação AxiomChangeExactlyToMax - ACEMa*

- **Objetivo:** Substituir o operando "Cardinality" pelo operando "maxCardinality".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *Cardinality* D), então essa relação é alterada para "C *maxCardinality* D". A relação passa a ser descrita na forma lógica apenas pelo operando "*maxCardinality*".
- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "C *Cardinality* D", o operando "*Cardinality*" é substituído por "*maxCardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*(temPessoaEnvolvida Cardinality 1 Paciente)*". Altera-se para "*(temPessoaEnvolvida maxCardinality 1 Paciente)*".

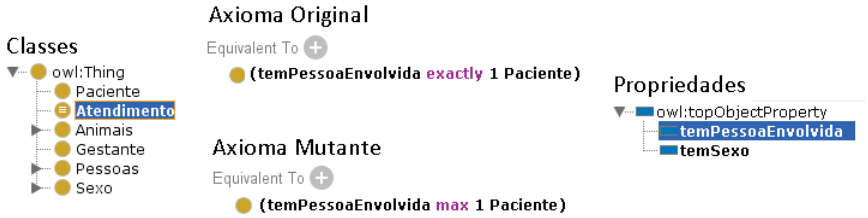


Figura 3.30: Modelo de mutação com o operador ACEMa (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação ACEMiMa (Subseção 3.4.2.9), devido à similaridade das mutações.

3.4.2.15 *Operador de mutação MaxCardAxiomUndef - MaCAU*

- **Objetivo:** Remover um axioma de cardinalidade do tipo "*maxCardinality*".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *maxCardinality* D), então essa relação *maxCardinaliy* é removida.
- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "C *maxCardinality* D", remove-se a expressão composta pelo operando "*maxCardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*Atendimento and (temPessoaEnvolvida maxCardinality 1 Paciente)*". Essa definição é alterada para "Atendimento".

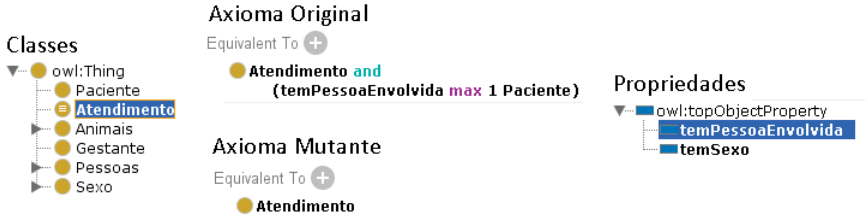


Figura 3.31: Modelo de mutação com o operador MaCAU (autor, 2019)

Os defeitos associados ao operador de mutação MaCAU são apresentados na Tabela 3.7.

Tabela 3.7: Defeitos associados ao operador de mutação MaCAU

Defeito	Motivo
Definição formal idêntica de classes e instâncias (Gómez-Pérez, 2004)	Classes ou indivíduos são descritos de forma idêntica com operadores de cardinalidade sem serem especificadas outras representações como a disjunção.
SumOfSomIsNeverEqualToOne (Corcho et al., 2009)	Uma classe é descrita incorretamente com o uso de uma restrição existencial e de cardinalidade simultaneamente.
SomeMeansAtLeastOne (Corcho et al., 2009)	A classe é descrita por uma restrição de cardinalidade que representa o mesmo conceito de outra restrição existencial.
Domain&CardinalityConstraints (Corcho et al., 2009)	A classe é descrita por uma restrição existencial e de cardinalidade em um único axioma, onde ambas as restrições definem o mesmo conceito.

3.4.2.16 Operador de mutação MinCardAxiomUndef - MiCAU

- **Objetivo:** Remover um axioma de cardinalidade do tipo "*minCardinality*".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *minCardinality* D), então essa relação *minCardinaliy* é removida.
- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma lógico como "C *minCardinality* D", o operador MiCAU altera esse axioma removendo a expressão composta pelo operando "*minCardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*Atendimento and (temPessoaEnvolvida minCardinality 1 Paciente)*". Essa definição é alterada para "Atendimento".

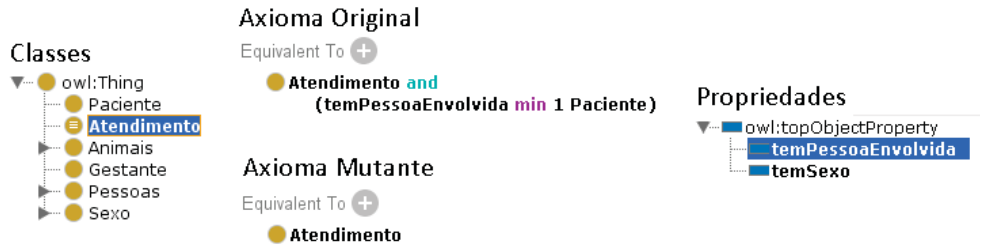


Figura 3.32: Modelo de mutação com o operador MiCAU (autor, 2019)

Os defeitos associados a este operador são os mesmos apresentados na Tabela 3.7. Para este operador também é associado o defeito *MinIsZero*, caso uma classe seja representada com uma restrição de cardinalidade mínima 0 (zero), o que não causa nenhum impacto na ontologia.

3.4.2.17 Operador de mutação ExaCardAxiomUndef - ECAU

- **Objetivo:** Remover um axioma de cardinalidade do tipo "*ExactCardinality*".
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *ExactCardinality* D), então essa relação *ExactCardinaliy* é removida.

- **Descrição:** Para uma classe descrita como subclasse ou equivalente por um axioma como "*C ExactCardinality D*", remove-se a expressão composta por "*ExactCardinality*".
- **Exemplo:** A classe Atendimento tem equivalência "*Atendimento and (temPessoaEnvolvida ExactCardinality 1 Paciente)*". Essa definição é alterada para "Atendimento".

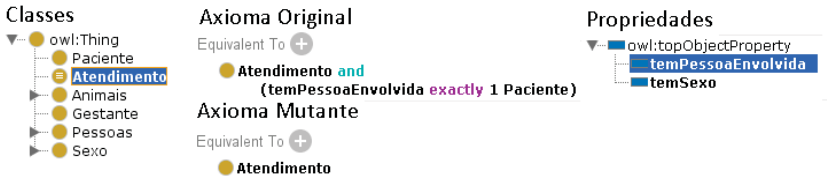


Figura 3.33: Modelo de mutação com o operador ECAU (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados na Tabela 3.7, devido à similaridade das mutações realizadas.

3.4.2.18 Operador de mutação UniversalAxiomUndef - UAU

- **Objetivo:** Remover um axioma universal em uma definição de classe.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *allValuesFrom* D), então essa relação C *allValuesFrom* D é removida.
- **Descrição:** Para uma classe primitiva ou definida descrita por um axioma universal como "*C allValuesFrom D*", remove-se a expressão composta por "*allValuesFrom*".
- **Exemplo:** A classe Atendimento possui as definições "*Atendimento and (temPessoaEnvolvida some Paciente)*" e "*Atendimento and (temPessoaEnvolvida only Paciente)*". O axioma "*Atendimento and (temPessoaEnvolvida only Paciente)*" é removido, permanecendo somente o axioma "*Atendimento and (temPessoaEnvolvida some Paciente)*", permitindo que outras instâncias participem dessa associação.

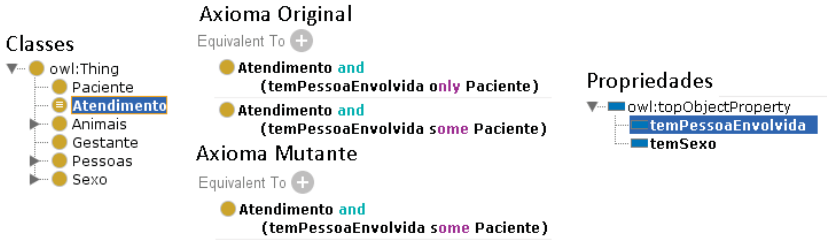


Figura 3.34: Modelo de mutação com o operador UAU (autor, 2019)

O operador de mutação UAU está associado aos seguintes defeitos:

- Definição formal idêntica de classes e instâncias (Gómez-Pérez, 2004).
- OnlynessIsLoneliness (Corcho et al., 2009).
- OnlynessIsLonelinessWithInheritance (Corcho et al., 2009).

- OnlynessIsLonelinessWithPropertyInheritance (Corcho et al., 2009).
- UniversalExistence (Corcho et al., 2009).
- UniversalExistenceWithInheritance1 (Corcho et al., 2009).
- UniversalExistenceWithInheritance2 (Corcho et al., 2009).
- UniversalExistenceWithPropertyInheritance (Corcho et al., 2009).
- UniversalExistenceWithInverseProperty (Corcho et al., 2009).
- SumOfSomWithPropertyInheritance (Corcho et al., 2009).
- SumOfSomWithInverseProperty (Corcho et al., 2009).

Todos esses defeitos, exceto para o defeito de definição formal idêntica de classes e instâncias, referem-se ao uso incorreto de restrições universais, que de um modo geral são abordados por Rector et al. (2004) e apresentados na Subseção 2.3.2 como uso equivocado de qualificadores de restrição. Desso modo, ao remover uma restrição universal, é possível a identificação desses defeitos caso o axioma removido tenha sido definido conforme esses padrões.

3.4.2.19 Operador de mutação ExistAxiomUndef - EAU

- **Objetivo:** Remover um axioma existencial em uma definição de classe.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada as classes C e D, tal que C e $D \in$ *OWLClass* e existe relação do tipo *equivalentClass* ou *subClassOf* entre C e D, relação esta descrita na forma lógica (C *someValueFrom* D), então essa relação *someValueFrom* é removida.
- **Descrição:** Para uma classe primitiva ou definida descrita por um axioma existencial como "*C someValuesFrom D*", remove-se a expressão composta por "*someValuesFrom*".
- **Exemplo:** A classe Atendimento possui as definições "*Atendimento and (temPessoaEnvolvida some Paciente)*" e "*Atendimento and (temPessoaEnvolvida only Paciente)*". O axioma existencial "*Atendimento and (temPessoaEnvolvida some Paciente)*" é removido, permanecendo somente o axioma universal "*Atendimento and (temPessoaEnvolvida only Paciente)*", permitindo que outras instâncias participem dessa associação.

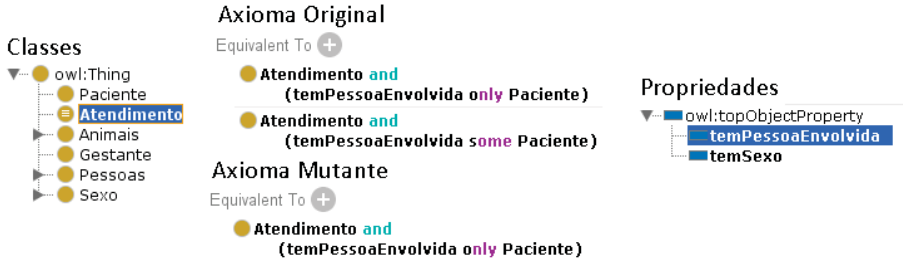


Figura 3.35: Modelo de mutação com o operador EAU (autor, 2019)

O operador de mutação EAU está associado aos seguintes defeitos:

- Definição formal idêntica de classes e instâncias (Gómez-Pérez, 2004).

- SumOfSom (Corcho et al., 2009).
- SumOfSomWithInheritage (Corcho et al., 2009).
- SumOfSomWithPropertyInheritance (Corcho et al., 2009).
- SumOfSomWithInverseProperty (Corcho et al., 2009).
- Domain&CardinalityConstraints (Corcho et al., 2009).

Todos esses defeitos, exceto para o defeito de definição formal idêntica de classes e instâncias, referem-se ao uso incorreto de restrições existenciais, que de um modo geral são abordados por Rector et al. (2004) e apresentados na Subseção 2.3.2 como uso equivocado de qualificadores de restrição. Ao remover uma restrição existencial, é possível a identificação desses defeitos caso o axioma removido tenha sido definido em conformidade com esses padrões.

3.4.2.20 Operador de mutação AxiomChangeTrueToFalse - ACTF

- **Objetivo:** Alterar para *false* a definição de uma propriedade de tipos de dados *booleana* definida como *true*.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada uma classe C e uma propriedade de tipos de dados P, tal que $C \in OWLClass$ e $P \in OWLDataProperty$ e existe relação do tipo *subClassOf* ou *EquivalentClass* entre C e P descrita na forma lógica (*P value true*), então essa relação é alterada para (*P value false*). A relação passa a ser descrita na forma lógica apenas pelo operando "*false*".
- **Descrição:** Para uma classe descrita por um axioma lógico como "*P value true*", o operador ACTF altera esse axioma substituindo o operando "*true*" pelo operando "*false*".
- **Exemplo:** A classe Atendimento possui o axioma de equivalência "*temPessoaEnvolvida value true*". Esse axioma é alterado para "*temPessoaEnvolvida value false*".

Figura 3.36: Modelo de mutação com o operador ACTF (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação ACMiMa (Subseção 3.4.2.9), devido à similaridade das mutações.

3.4.2.21 Operador de mutação AxiomChangeFalseToTrue - ACFT

- **Objetivo:** Alterar para *true* a definição de uma propriedade de tipos de dados *booleana* definida como *false*.

- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, dada uma classe *C* e uma propriedade de tipos de dados *P*, tal que *C* ∈ *OWLClass* e *P in OWLDataProperty* e existe relação do tipo *subClassOf* ou *EquivalentClass* entre *C* e *P* descrita na forma lógica (*P value false*), então essa relação é alterada para (*P value true*). A relação passa a ser descrita na forma lógica apenas pelo operando "*true*".
- **Descrição:** Para uma classe descrita por um axioma como "*P value false*", o operador ACFT altera esse axioma substituindo o operando "*false*" pelo operando "*true*".
- **Exemplo:** A classe Atendimento possui o axioma de equivalência "*temPessoaEnvolvida value false*". Esse axioma é alterado para "*temPessoaEnvolvida value true*".

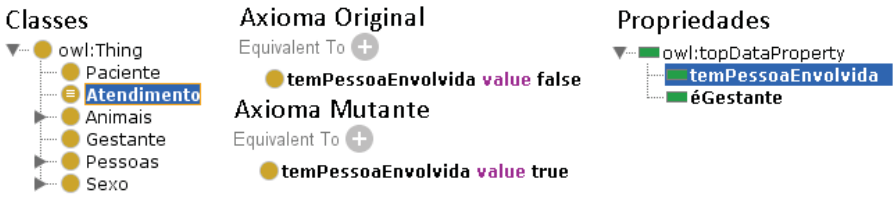


Figura 3.37: Modelo de mutação com o operador ACFT (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação ACMiMa (Subseção 3.4.2.9), devido à similaridade das mutações.

3.4.2.22 Operador de mutação ChangeIndividualToSubClass - CIS

- **Objetivo:** Alterar para *SubClassOf* a definição de um indivíduo.
- **Definição:** Considerando o conjunto de classes *OWLClass* e o conjunto de indivíduos *Individuals* do metamodelo *OWL*, dada uma classe *C* e um indivíduo *I*, tal que *C* ∈ *OWLClass* e *I* ∈ *Individuals* e existe relação *InstanceOf* entre *I* e *C* (*I* é uma instância de *C*), então essa relação é alterada para "*SubClassOf*", *I* passa a ser subclasse de *C*.
- **Descrição:** Para um indivíduo descrito por um axioma lógico "*I InstanceOf C*", o operador CIS altera esse axioma substituindo o operador "*InstanceOf*" por "*SubClassOf*".
- **Exemplo:** A classe *Pessoas* possui como instância o indivíduo *Homem*. Esse indivíduo é alterado para subclasse de *Pessoas*.

Figura 3.38: Modelo de mutação com o operador CIS (autor, 2019)

Os defeitos associados a este operador de mutação são o "alto grau de especialização" (Poveda-Villalón et al., 2010) e "utilizar elementos incorretamente" (Poveda-Villalón et al., 2010), dado um nó da ontologia não ser instanciado por ser um indivíduo.

3.4.2.23 *Operador de mutação IndividualDisjointUndef - IDU*

- **Objetivo:** Remover a definição de disjunção entre um grupo de indivíduos.
- **Definição:** Considerando o conjunto de indivíduos *Individuals* do metamodelo *OWL*, dado os indivíduos I_1 e I_2, tal que I_1 e $I_2 \in$ *Individuals* e existe relação *DifferentIndividuals* entre I_1 e I_2 (I_1 é disjunto de I_2), essa relação *DifferentIndividuals* é removida do modelo.
- **Descrição:** Para um conjunto de indivíduos descritos por uma relação "*DifferentIndividuals*", o operador IDU remove essa relação.
- **Exemplo:** A classe *Pessoas* possui como instâncias os indivíduos *Homem* e *Mulher*. Esses indivíduos estão definidos como disjuntos entre si. Essa relação de disjunção entre os indivíduos *Homem* e *Mulher* é removida, o que influencia diretamente a classificação de indivíduos na classe *Pessoas* ou em outras classes associadas a classe *Pessoas*.

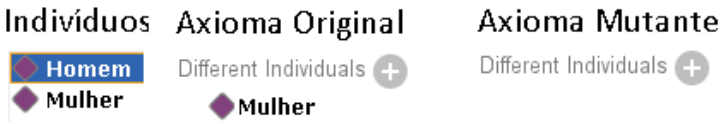


Figura 3.39: Modelo de mutação com o operador IDU (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação CIS (Subseção 3.4.2.22). Esses defeitos podem ser revelados caso a definição de disjunção entre os indivíduos tenha sido feita de forma equivocada ou após a remoção da disjunção um indivíduo não possa ser instanciado dada a existência do defeito de alto grau de especialização, onde esse indivíduo deveria ser definido como uma classe.

3.4.2.24 *Operador de mutação IndividualSameUndef - ISU*

- **Objetivo:** Remover a definição de equivalência entre um grupo de indivíduos.
- **Definição:** Considerando o conjunto de indivíduos *Individuals* do metamodelo *OWL*, dado os indivíduos I_1 e I_2, tal que I_1 e $I_2 \in$ *Individuals* e existe relação *SameIndividual* entre I_1 e I_2 (I_1 é o mesmo indivíduo que I_2), essa relação *SameIndividual* é removida.
- **Descrição:** Para um conjunto de indivíduos descritos por uma relação "*SameIndividual*", o operador ISU remove essa relação.
- **Exemplo:** A classe *Pessoas* possui como instâncias os indivíduos *Adulto* e *Idoso*. Esses indivíduos estão definidos como equivalentes entre si. Essa relação de equivalência entre os indivíduos *Adulto* e *Idoso* é removida, o que influencia diretamente a classificação de indivíduos na classe *Pessoas* ou em outras classes associadas a classe *Pessoas*.

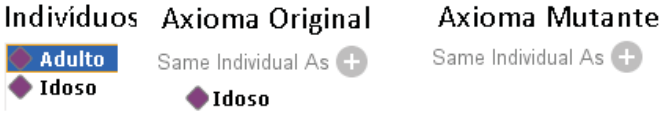


Figura 3.40: Modelo de mutação com o operador ISU (autor, 2019)

Os defeitos associados a este operador de mutação são os mesmos defeitos apresentados para o operador de mutação IDU (Subseção 3.4.2.23), devido à similaridade das mutações realizadas por estes dois operadores.

3.5 CONSIDERAÇÕES DO CAPÍTULO

As definições apresentadas na Seção 3.1 definem a caracterização dos tipos de defeitos abordados pelo teste de mutação para ontologias, abrangendo os tipos de defeitos apresentados na Subseção 2.3.2, que auxiliaram a construção dos operadores de mutação. Na Seção 3.2 foram apresentas as definições de dados e casos de teste, que fundamentam o método para a geração dos dados de teste definido nesta tese, que é apresentado na Seção 3.3.1. Na Seção 3.3 foi apresentada uma abordagem conceitual do teste de mutação para ontologias OWL, com relação à geração dos mutantes, geração dos dados de teste e a aplicação do teste de mutação, sendo cada uma destas etapas expandida em mais detalhes na Subseção 3.3.1 e no Capítulo 4 com uma exemplificação da aplicação desta técnica de teste.

Na Seção 3.4 foram apresentados e definidos os 38 operadores de mutação propostos nesta tese. Ao analisar os operadores de mutação já existentes (Porn, 2014), 17 operadores foram reaproveitados e classificados como operadores de mutação estrutural e operadores lógicos de mutação, 4 deram origem a 5 novos operadores de mutação e foram desconsiderados junto com outros 4 operadores restantes. A Tabela 3.8 a seguir apresenta de forma resumida os 38 operadores de mutação divididos em 14 operadores de mutação estrutural e 24 operadores lógicos de mutação, e também como foram reaproveitados e desconsiderados os operadores já existentes. Todos os operadores foram implementados na ferramenta MutaOnto, que é apresentada Seção 4.5, para a geração das ontologias mutantes apresentada na Seção 4.1.

Tabela 3.8: Classificação dos operadores de mutação estrutural e operadores lógicos de mutação (autor, 2019)

Operadores	**Classificação**	**Seção**	**Op. de Mutação** (Porn, 2014)	**Descontinuado**
CUC	Estrutural	3.4.1.1	CUC	
CDD	Estrutural	3.4.1.2	CDD	
CEU	Estrutural	3.4.1.3	CEU	
PDD	Estrutural	3.4.1.4	PDD	
PDUP	Estrutural	3.4.1.5	PDUP	
PRD	Estrutural	3.4.1.6	PRD	
PRUP	Estrutural	3.4.1.7	PRUP	
PDU	Estrutural	3.4.1.8	PDU	
PRU	Estrutural	3.4.1.9	PRU	
ACSD	Estrutural	3.4.1.10	CDDO	CDDO
ACED	Estrutural	3.4.1.11	CDDO	CDDO
PDC	Estrutural	3.4.1.12	CED	CED
DPC	Estrutural	3.4.1.13	CED	CED
CDU	Estrutural	3.4.1.14	CDUO CDUA	CDUO CDUA
CEUA	Lógico	3.4.2.1	CEUA	
CEUO	Lógico	3.4.2.2	CEUO	
ACATO	Lógico	3.4.2.3	ACATO	
ACOTA	Lógico	3.4.2.4	ACOTA	

Continua na próxima página

Tabela 3.8 – *Continua da página anterior*

Operadores	**Classificação**	**Seção**	**Op. de Mutação** (Porn, 2014)	**Descontinuado**
ACATS	Lógico	3.4.2.5	ACATS	
ACSTA	Lógico	3.4.2.6	ACSTA	
AEDN	Lógico	3.4.2.7	AEDN	
AEUN	Lógico	3.4.2.8	AEUN	
ACMiMa	Lógico	3.4.2.9	Não existe	
ACMaMi	Lógico	3.4.2.10	Não existe	
ACMiEx	Lógico	3.4.2.11	Não existe	
ACMaEx	Lógico	3.4.2.12	Não existe	
ACEMi	Lógico	3.4.2.13	Não existe	
ACEMa	Lógico	3.4.2.14	Não existe	
MacAU	Lógico	3.4.2.15	Não existe	
MiCAU	Lógico	3.4.2.16	Não existe	
ECAU	Lógico	3.4.2.17	Não existe	
UAU	Lógico	3.4.2.18	Não existe	
EAU	Lógico	3.4.2.19	Não existe	
ACTF	Lógico	3.4.2.20	Não existe	
ACFT	Lógico	3.4.2.21	Não existe	
CIS	Lógico	3.4.2.22	Não existe	
IDU	Lógico	3.4.2.23	Não existe	
ISU	Lógico	3.4.2.24	Não existe	
			CECO	CECO
			CUP	CUP
			PID	PID
			PIUD	PIUD

4 APLICAÇÃO DO TESTE DE MUTAÇÃO PARA ONTOLOGIAS OWL

Neste capítulo é apresentada uma exemplificação da aplicação do teste de mutação para ontologias OWL de acordo com as definições apresentadas na Seção 3.3 e Subseção 3.3.1. Nas Seções 4.1 e 4.2 a seguir, é apresentado um exemplo do processo de geração dos mutantes e de dados de teste que são necessários para a execução dos testes da ontologia original e das ontologias mutantes (Seção 4.3). Na sequência é apresentado um exemplo da análise dos mutantes na Seção 4.4. Para automatizar o processo de geração dos mutantes e de dados de teste, na Seção 4.5 é apresentada a ferramenta MutaOnto desenvolvida para essa atividade. Na Subseção 4.5.1 é apresentada a arquitetura desta ferramenta e na Subseção 4.5.2 uma síntese de seu funcionamento.

4.1 GERAÇÃO DAS ONTOLOGIAS MUTANTES

Para fins de ilustração da Definição III da Subseção 3.3.1, o Exemplo 4.1 apresenta um axioma original que descreve a classe *QuietDestination* da ontologia *travel* (Figura 2.6). No Exemplo 4.2 é apresentado o axioma do Exemplo 4.1 mutado com o operador lógico de mutação ACATO (Subseção 3.4.2). De acordo com a Definição III, ao aplicar o operador de mutação ACATO no axioma original apresentado no Exemplo 4.1, será gerada uma nova ontologia mutante para este operador, dado que existe somente um operador OWL "*ObjectIntersectionOf*" (AND lógico) neste axioma, que será substituído pelo operador OWL "*ObjectUnionOf*" (OR lógico). Um axioma original refere-se a um axioma definido na ontologia em teste que ainda não sofreu nenhum tipo de mutação, já um axioma mutante ou mutado refere-se ao axioma original com alguma alteração sintática produzida por um operador de mutação.

Exemplo 4.1: Axioma original que descreve a classe QuietDestination da ontologia travel (autor, 2019)

```
- AXIOMA ORIGINAL DA ONTOLOGIA TRAVEL
  <EquivalentClasses>
      <Class IRI="#QuietDestination"/>
      <ObjectIntersectionOf>
          <Class IRI="#Destination"/>
          <ObjectComplementOf>
              <Class IRI="#FamilyDestination"/>
          </ObjectComplementOf>
      </ObjectIntersectionOf>
  </EquivalentClasses>
```

Exemplo 4.2: Axioma do Exemplo 4.1 mutado com o operador lógico de mutação ACATO (autor, 2019)

```
- AXIOMA DE UMA ONTOLOGIA MUTANTE DA ONTOLOGIA TRAVEL
  <EquivalentClasses>
      <Class IRI="#QuietDestination"/>
      <ObjectUnionOf>
          <Class IRI="#Destination"/>
          <ObjectComplementOf>
              <Class IRI="#FamilyDestination"/>
          </ObjectComplementOf>
      </ObjectUnionOf>
  </EquivalentClasses>
```

4.2 GERAÇÃO DE DADOS DE TESTE

Para fins de ilustração da geração de um dado de teste, o Quadro 4.1 apresenta um exemplo da geração de dados de teste com o operador lógico de mutação ACATO, apresentado na Subseção 3.4.2 e aplicado no axioma do Exemplo 3.2 da Seção 3.2, que refere-se à descrição da classe *BudgetAccommodation* da ontologia *travel* da Figura 2.6.

Quadro 4.1: Dados de teste gerados a partir do operador de mutação ACATO (autor, 2019)

Classe	**Axioma original *A***	**Axioma mutante *A'***
BudgetAccommodation	*Accommodation* $\sqcap$ ($\exists$*hasRating*.(*oneStarRating*, *twoStarRating*))	*Accommodation* $\sqcup$ ($\exists$*hasRating*.(*oneStarRating*, *twoStarRating*))

Conforme o exemplo apresentado no Quadro 4.1, são gerados dois dados de teste, sendo um dado de teste o axioma original da ontologia e o outro o axioma mutante gerado a partir da mutação realizada pelo operador de mutação ACATO. Como neste axioma existe somente um operando "AND" para ser substituído pelo operando "OR" com o operador de mutação ACATO, é possível gerar somente um dado de teste a partir do axioma original. O dado de teste do Quadro 4.1 é apresentado no Quadro 4.2 formalizado na sintaxe *OWL Manchester*, que é compatível com a ferramenta *Protégé* para a aplicação dos dados durante os testes das ontologias.

Quadro 4.2: Dados de teste gerados a partir do operador de mutação ACATO em *OWL Manchester* (autor, 2019)

Classe	**Axioma original *A***	**Axioma mutante *A'***
BudgetAccommodation	*Accommodation and* (*hasRating some* {*oneStarRating*, *twoStarRating*})	*Accommodation or* (*hasRating some* {*oneStarRating*, *twoStarRating*})

De acordo com os dados de teste do Quadro 4.2, para realizar a seleção dos axiomas lógicos tanto originais quanto mutantes da ontologia em teste para a composição de *T*, é necessário que essa ontologia esteja representada na sintaxe *OWL Manchester*. A ferramenta *Protégé* possibilita converter uma ontologia OWL para *OWL Manchester* assim como o processo inverso. Assim, os dados de teste podem ser gerados a partir de qualquer ontologia OWL.

Os dados de teste gerados a partir dos operadores de mutação correspondem a um dos tipos de lógica descritiva apresentados no Quadro 2.4. Esses tipos de lógica descritiva correspondem ao conjunto de construtores OWL que sofrem mutações a partir de um grupo específico de operadores lógicos de mutação. Esse grupo de operadores de mutação utilizado na geração dos dados de teste é composto pelos operadores ACATO, ACOTA, CEUA, CEUO, AEDN, AEUN, ACATS, ACSTA, ACMiMa, ACMaMi, ACMiEx, ACMaEx, ACEMi, ACEMa, ACTF, ACFT e PDC. Do total de 38 operadores de mutação para ontologias OWL, 17 operadores são utilizados no processo automático de geração dos dados de teste. Cada dado de teste gerado está associado a no mínimo um dos tipos de lógica desritiva $\mathcal{AL}$, $\mathcal{U}$, $\mathcal{C}$, $\mathcal{E}$, $\mathcal{N}$ e $\mathcal{O}$, formando um conjunto de dados de teste composto pela lógica descritiva $\mathcal{ALUCENO}$. Assim, para cada mutação realizada por um dos operadores de mutação que geram dados de teste em um axioma de *O*, o axioma original e todos os axiomas mutantes gerados são selecionados, dando origem ao conjunto de dados de teste *T*, conforme a definição I da Seção 3.2.

4.3 TESTE DAS ONTOLOGIAS

Após a definição de T, cada $t \in T$ é aplicado em O com o auxílio de uma ferramenta para realizar consultas em ontologias OWL. Cada t é interpretado por um classificador que analisa o dado de teste de acordo com os axiomas definidos em O, retornando um resultado com base na instanciação de classes e indivíduos encontrados. A Figura 4.1 apresenta um exemplo do resultado da aplicação de t do axioma original do Quadro 4.2 na ontologia *travel* (Figura 2.6).

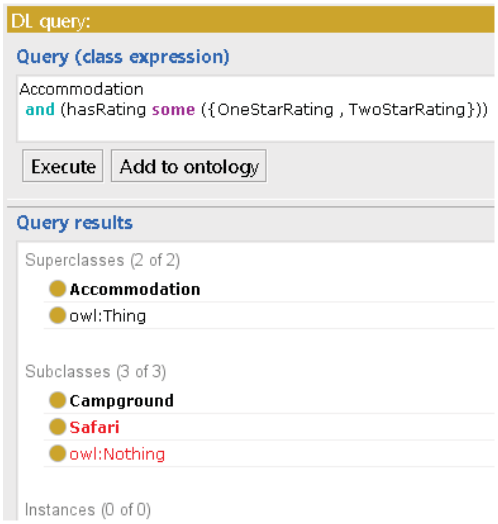


Figura 4.1: Resultado da aplicação de um dado de teste na ontologia original (*travel*) usando o *Protégé* (autor, 2019)

O testador é o responsável pela análise do resultado obtido. Caso o resultado seja identificado como correto, é utilizado como resultado esperado para a comparação com os resultados obtidos pela aplicação do mesmo dado de teste nas ontologias mutantes.

O próximo passo consiste em aplicar o conjunto de dados de teste T aplicado em O em cada uma das ontologias mutantes $o' \in O'$ geradas pelos operadores de mutação. A Figura 4.2 apresenta um exemplo com o resultado da aplicação do mesmo dado de teste da Figura 4.1, na ontologia *travel*, onde o axioma que define a classe *BudgetAccommodation* foi mutado com o operador de mutação ACATO, substituindo o operando *and* pelo operando *or* nesse axioma dando origem a uma nova ontologia mutante.

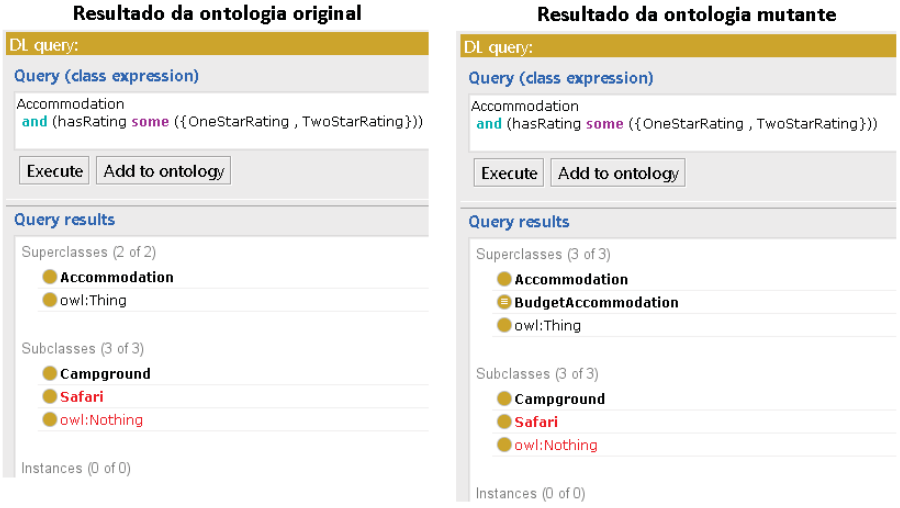


Figura 4.2: Análise dos resultados (autor, 2019)

Conforme a Figura 4.2, após a aplicação do dado de teste na ontologia original e na ontologia mutante, é possível observar que na ontologia mutante a classe *BudgetAccommodation* aparece como superclasse dela mesma. Deste modo, esse mutante é considerado morto, indicando que o dado de teste identificou o defeito inserido pelo operador de mutação.

4.4 ANÁLISE DOS MUTANTES

Após a aplicação de todos os dados de teste do conjunto de dados de teste *T*, cada mutante vivo é analisado individualmente pelo testador e considerado equivalente caso não sejam encontrados defeitos, caso contrário, o defeito é corrigido e os dados de teste são novamente aplicados com o objetivo de matar o mutante. Caso o mutante não seja morto, é considerado equivalente à ontologia original. Um mutante equivalente corresponde a uma ontologia mutante que produz os mesmos resultados que a ontologia original para qualquer dado de teste aceito pela ontologia. Conforme Delamaro et al. (2007), decidir se os mutantes vivos são equivalentes à *O* é uma questão indecidível, ficando a critério do testador a realização da tarefa de oráculo na análise dos mutantes (Figura 3.2) para definí-los como equivalentes a *O* ou caso contrário, a revelação de um defeito em *O*.

4.5 FERRAMENTA MUTAONTO

A ferramenta MutaOnto[1] (Mutação de Ontologias) foi desenvolvida nesta tese para auxiliar o testador a gerar de forma automática os mutantes e os dados de teste com base em mutações estruturais e mutações lógicas. Esta ferramenta é composta pelos dois conjuntos de operadores de mutação para ontologias OWL apresentados nas Subseções 3.4.1 e 3.4.2. Assim, evita-se que enganos humanos sejam cometidos na geração dos mutantes e dos dados de teste através de um processo manual, como por exemplo, deixar de realizar mutações em descrições lógicas ou gerar mutantes duplicados.

A ferramenta MutaOnto consiste em um programa desenvolvido na linguagem de programação *Object Pascal*, utilizando o Ambiente de Desenvolvimento Integrado - IDE (do inglês, *Integrated Development Environment*) Delphi 10.3, que precisa ser executado localmente em um computador. Esta ferramenta pode ser utilizada para qualquer ontologia OWL, sendo independente de domínio ou ambiente de desenvolvimento de ontologias. Nesta seção é apresentada a arquitetura da ferramenta MutaOnto (Subseção 4.5.1), sendo abordada a organização interna e seus principais componentes. Em seguida, na Subseção 4.5.2, é apresentado um exemplo do funcionamento da ferramenta para auxílio ao testador durante a execução do teste de mutação, seguido pela exemplificação da geração dos mutantes (Subseção 4.1) e geração de dados de teste (Subseção 4.2) pela ferramenta MutaOnto.

4.5.1 Arquitetura da ferramenta

A Figura 4.3 apresenta a arquitetura da ferramenta MutaOnto. A ferramenta toma como entrada um arquivo OWL de uma ontologia a ser testada. Em seguida, na etapa "*1) Seleção dos operadores de mutação*", são selecionados os operadores de mutação que serão utilizados nas etapas "*2.a) Geração dos mutantes O'*" e "*2.b) Geração dos dados de teste T*", para a geração do conjunto de mutantes *O'* e do conjunto de dados de teste *T* que serão utilizados na execução do teste da ontologia original e das ontologias mutantes.

[1]Disponível em: https://github.com/alexporn/MutaOnto

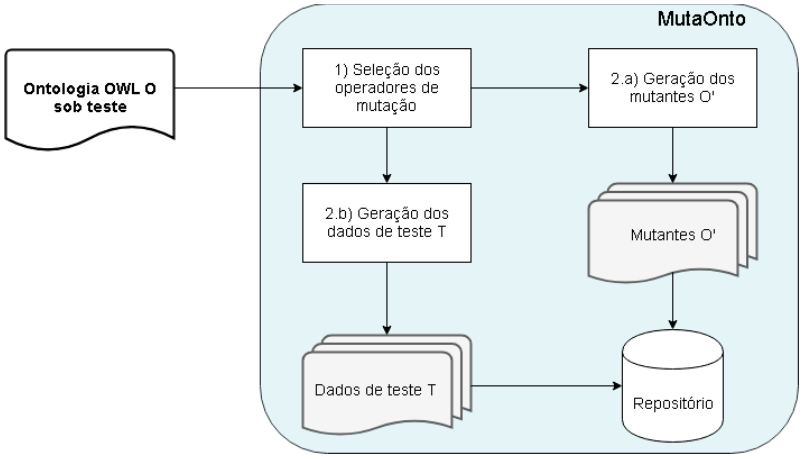


Figura 4.3: Arquitetura da ferramenta MutaOnto (autor, 2019)

Os mutantes *O'*, gerados na etapa "*2.a) Geração dos mutantes O'*" da Figura 4.3, são armazenados separadamente em repositórios de acordo com o operador de mutação que os gerou. A Figura 4.4 A) apresenta um exemplo do repositório dos mutantes e a Figura 4.4 B) um exemplo dos mutantes gerados com o operador de mutação estrutural CUC da Subseção 3.4.1.1.

A) Repositório dos mutantes

Este Computador › OS (C:) › Teste Mutação Ontologias › CDAO

Nome	Data de modificaç...
ACATO	10/01/2019 17:59
ACED	10/01/2019 17:59
ACFT	10/01/2019 17:59
ACOTA	10/01/2019 17:59
ACSD	10/01/2019 17:59
ACTF	10/01/2019 17:59

B) Mutantes do operador de mutação CUC

Este Computador › OS (C:) › Teste Mutação Ontologias › CDAO › CUC

Nome	Data de modificaç...	Tipo	Tamanho
cuc13923.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc14501.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc15271.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc16298.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc16614.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc17381.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc18171.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc18973.owl	10/01/2019 17:59	Arquivo OWL	90 KB
cuc19763.owl	10/01/2019 17:59	Arquivo OWL	90 KB

Figura 4.4: Exemplo do repositório de mutantes (autor, 2019)

Com relação ao conjunto de dados de teste *T*, gerados na etapa "*2.b) Geração dos dados de teste T*" da Figura 4.3, é criado automaticamente um arquivo texto contendo todas as variantes dos axiomas mutados de acordo com os operadores de mutação selecionados. A Figura 4.5 apresenta um exemplo de um conjunto de dados de teste *T* gerado pelo processo da etapa "*2.b) Geração dos dados de teste T*" da Figura 4.3.

Conjunto de dados de teste

```
        éPontoTuristicode some Canterbury
-------------------------------------------------
        temPontoTuristico some St._Paul's_Cathedral
-------------------------------------------------
        éCidadeDe some Costa_do_Marfim
-------------------------------------------------
        temPais some Africa_do_Sul
-------------------------------------------------
        temPais some Angola
-------------------------------------------------
        temPais some Costa_do_Marfim
```

Figura 4.5: Exemplo de um conjunto de dados de teste *T* (autor, 2019)

4.5.2 Síntese de funcionamento

A Figura 4.6 apresenta a interface gráfica da ferramenta MutaOnto para a geração dos mutantes e dos dados de teste. A interface com o testador consiste primeiramente no carregamento de um arquivo OWL contendo o código fonte de uma ontologia a ser testada, através do botão "*Carregar Ontologia*", como indicado no Passo 1 destacado na Figura 4.6. O código fonte da ontologia é apresentado na lateral esquerda da ferramenta.

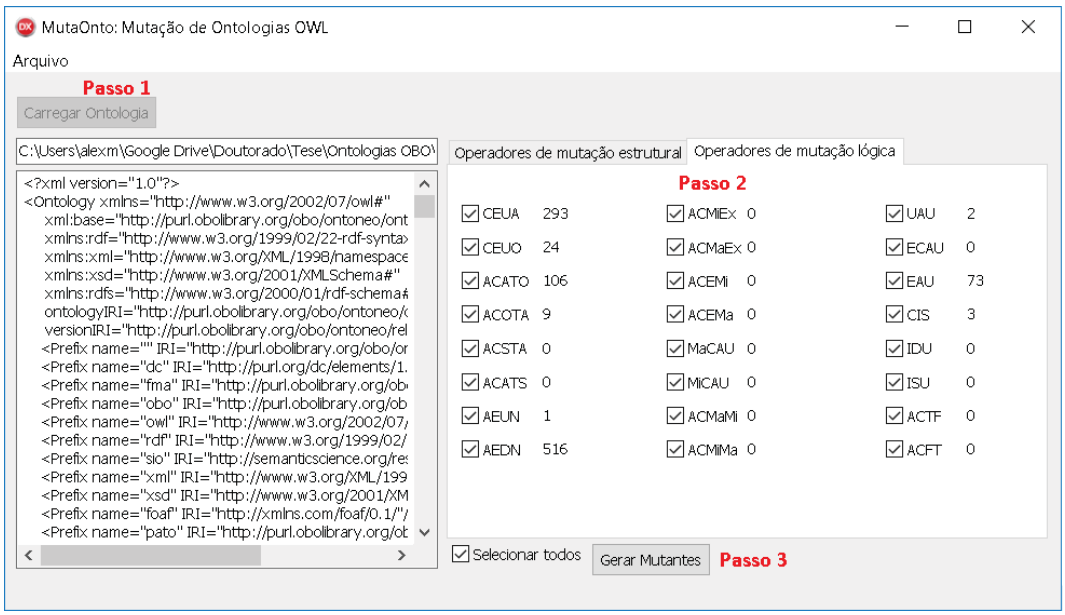


Figura 4.6: Interface gráfica da ferramenta MutaOnto (autor, 2019)

Com o arquivo OWL carregado na ferramenta, o próximo passo consiste na seleção dos operadores de mutação a serem aplicados nesta ontologia OWL *O*, como indicado no Passo 2 da Figura 4.6. Os operadores de mutação são apresentados no lado direito da ferramenta, sendo separados pelos conjuntos *OME* e *OLM* das Subseções 3.4.2 e 3.4.1, representados por mutação estrutural e mutação lógica. O testador tem a possibilidade de selecionar todos os operadores dos dois grupos, marcando a caixa de seleção "*Selecionar todos*", ou pode selecionar individualmente cada operador de mutação que pretenda utilizar.

Em seguida, através do botão "*Gerar Mutantes*", como indicado no Passo 3 da Figura 4.6, de acordo com os operadores de mutação selecionados, é realizada a geração do conjunto de mutantes *O'*. Para a geração do conjunto de dados de teste *T*, primeiramente deve ser carregada na ferramenta MutaOnto a mesma ontologia em formato *OWL Manchester* através do Passo 1 da Figura 4.6. Em seguida, ainda com os operadores de mutação selecionados, no menu "*Arquivo*" contém a função "*Gerar dados de teste*". Com um clique nessa função será gerado automaticamente um conjunto de dados de teste *T*, conforme mostrado no exemplo da Figura 4.5, para ser aplicado na ontologia original *O* e em cada ontologia mutante om_{ij} gerada.

4.6 CONSIDERAÇÕES DO CAPÍTULO

Conforme apresentado na Seção 4.2, para gerar automaticamente o conjunto de dados de teste *T* utilizando a ferramenta *MutaOnto*, é necessário que a ontologia OWL esteja representada na sintaxe *OWL Manchester*. Os operadores de mutação que são utilizados para gerar os dados de teste, não convertem os axiomas lógico-descritivos da ontologia OWL para *OWL Manchester*. Estes operadores realizam mutações nas duas sintaxes, de acordo com a sintaxe da ontologia

que foi carregada na ferramenta *MutaOnto*. Deste modo, após a geração de todos os mutantes, é necessário utilizar uma ferramenta de auxílio, como a ferramenta *Protégé*, para realizar a conversão da sintaxe da ontologia original OWL para *OWL Manchester* e então gerar os dados de teste automaticamente.

Para a execução do teste de mutação, é utilizada a ferramenta *Protégé* para a aplicação dos dados de teste, por ser uma ferramenta de referência para a construção e edição de ontologias OWL. Esta ferramenta, além de ser composta por um editor de ontologias, possui uma biblioteca de módulos de extensão com diversas funcionalidades, como avaliação, comparação e mapeamento de ontologias, assim como possibilita a utilização de classificadores para análise de inferências lógicas. O *Protégé* disponibiliza uma série de classificadores, dentre eles o classificador *Pellet*. Conforme Abburu (2012), Dentler et al. (2011) e Sirin et al. (2007), foi o primeiro classificador a suportar todos os elementos da lógica descritiva em OWL, além de juntamente com o classificador *ELK*, serem os únicos a suportarem classificação incremental.

Considera-se para a ferramenta *MutaOnto* futuras atualizações como a implementação de um classificador para que seja possível a aplicação dos dados de teste diretamente na ferramenta *MutaOnto*, podendo desconsiderar o uso do *Protégé*. Isto permitirá também que cada dado de teste aplicado na ontologia original O que mate o mutante om_{ij}, seja armazenado separadamente, gerando assim um conjunto de dados de teste minimizado, para aplicações futuras em outras técnicas de teste, como o teste de regressão (Pressman, 2011). Também considera-se a expansão dos operadores de mutação para outras sintaxes da linguagem OWL, como por exemplo, expandir todos os operadores para realizar mutações em *OWL Manchester* e não somente os operadores utilizados para gerar os dados de teste.

5 AVALIAÇÃO DO TESTE DE MUTAÇÃO PARA ONTOLOGIAS OWL

Com o objetivo de avaliar a abordagem do teste de mutação para ontologias OWL proposta nesta tese, foram conduzidos dois experimentos para investigar a aplicabilidade dos operadores de mutação e dos dados de teste em identificar os defeitos simulados pelos operadores de mutação e revelar possíveis defeitos cometidos durante o desenvolvimento de ontologias. Em cada um dos experimentos não houve participação deste pesquisador no desenvolvimento das ontologias, de modo que não se tem conhecimento inicial da existência de defeitos em cada ontologia testada. No Experimento I são consideradas ontologias desenvolvidas sem o uso de métodos ou práticas para o desenvolvimento e avaliação de ontologias[1] e, no Experimento II são consideradas ontologias desenvolvidas por especialistas que utilizam princípios que especificam as melhores práticas no desenvolvimento[2] (Haendel et al., 2019). Com base nos resultados obtidos nestes dois cenários, é possível avaliar o teste de mutação para ontologias considerando ontologias construídas por desenvolvedores aprendizes e ontologias construídas por desenvolvedores especialistas. Dada a experiência dos desenvolvedores, essas ontologias podem apresentar menos defeitos do que as ontologias construídas pelos aprendizes. Nas Seções 5.1 e 5.2 são apresentados os experimentos, bem como as motivações, objetivos, métodos e os resultados obtidos em cada um.

5.1 EXPERIMENTO I

Neste experimento é proposto avaliar o método do teste de mutação para ontologias OWL que foram construídas sem o uso de métodos e técnicas de desenvolvimento propostos na literatura, uma vez que estes métodos não são considerados nesta tese para a aplicação do teste de mutação em ontologias OWL.

5.1.1 Motivação

Conforme Hendler (2001), ontologias pequenas e altamente contextualizadas para diferentes domínios são desenvolvidas localmente por diferentes profissionais e não por especialistas em ontologias. Essa diversidade de perfis de desenvolvedores pode facilitar a ocorrência de defeitos pelo desconhecimento de padrões de enganos e modelagem de ontologias. Assim, pretende-se com este experimento analisar as seguintes questões de pesquisa:

1. É possível revelar os defeitos apresentados na Subseção 2.3.2 utilizando o conjunto de operadores de mutação propostos nesta tese?

2. É possível que um operador de mutação revele um defeito diferente ao qual ele está associado?

3. O teste de mutação proposto é capaz de revelar outros tipos de defeitos que não constam na lista de defeitos da Subseção 2.3.2?

4. Os dados de teste gerados pelo método de geração de dados de teste proposto são capazes de identificar os defeitos simulados pelos operadores de mutação?

[1]Ontologias disponíveis em: https://github.com/alexporn/Ontologias-Experimento-I
[2]Ontologias disponíveis em: https://github.com/alexporn/Ontologias-Experimento-II

- **Objetivo**
  - Avaliar o método de teste de mutação para ontologias OWL em ontologias que foram construídas sem o uso de métodos e técnicas de desenvolvimento propostos na literatura, para avaliar a identificação de defeitos e adequação dos dados de teste.

5.1.2 Método

O Experimento I é baseado em quatro atividades distintas: (i) treinamento em desenvolvimento de ontologias OWL para uma turma de 30 alunos com faixa etária de 16 a 18 anos de idade, de um curso Técnico em Informática Integrado ao Ensino Médio, que já cursaram ou estejam cursando as disciplinas de Engenharia de Software e Bancos de Dados. Estas disciplinas são fundamentais para que os participantes já possuam conhecimentos no desenvolvimento de sistemas. A segunda atividade corresponde em (ii) dividir a turma em grupos de no máximo 5 alunos, totalizando 6 grupos, onde cada grupo desenvolveu uma ontologia de um domínio específico do conhecimento, (iii) cada ontologia foi testada utilizando o método do teste de mutação para ontologias OWL proposto nesta tese. A última atividade consiste em (iv) avaliar os resultados obtidos após a aplicação do teste, com relação ao número de mutantes mortos e vivos, o que possibilita avaliar a existência de defeitos e os seus tipos e a equivalência entre os mutantes e a ontologia original. A Figura 5.1 apresenta a sequência das atividades. Conforme pode ser observado, na atividade (ii) não houve interação com os alunos para o desenvolvimento das ontologias, somente eles realizaram esta atividade após o término do treinamento, de modo que não é possível a identificação dos defeitos antes da aplicação dos testes.

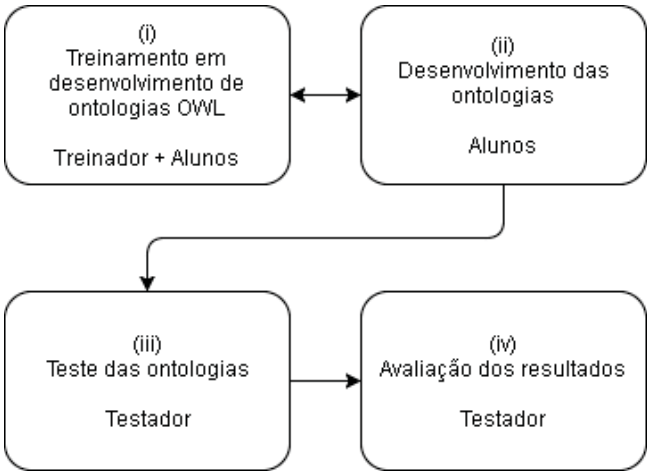


Figura 5.1: Metodologia do Experimento I (autor, 2019)

Cada atividade apresentada na Figura 5.1 possui os seguintes objetivos:

- Atividade (i): a atividade de treinamento em desenvolvimento de ontologias OWL para alunos de um curso Técnico em Informática Integrado ao Ensino Médio, tem como objetivo a criação das ontologias por um grupo de desenvolvedores que não são especialistas em ontologias e, por outro lado, desconhecem métodos e práticas de desenvolvimento e avaliação adequados. Esta atividade justifica-se por estar alinhada com os objetivos deste experimento, diante do exposto por Hendler (2001), de que ontologias pequenas e altamente contextualizadas para diferentes domínios são desenvolvidas localmente por diferentes profissionais e não por especialistas em ontologias.

- Atividade (ii): esta atividade tem como objetivo obter um conjunto de ontologias de vários domínios do conhecimento. Conforme representado na Figura 5.1, os alunos podem tirar dúvidas durante o desenvolvimento com o treinador, porém o treinador não interfere no modo de desenvolvimento dos alunos. Esta atividade justifica-se pelo fato de possibilitar avaliar a aplicação dos operadores de mutação e dos dados de teste em ontologias desenvolvidas por desenvolvedores não especialistas e que abordam diferentes domínios do conhecimento.

- Atividade (iii): esta atividade tem como objetivo validar os operadores de mutação, a ferramenta para geração dos mutantes e dados de teste e o método de teste de mutação para ontologias OWL proposto nesta tese. Esta atividade é realizada apenas pelo testador, dado que o objetivo do teste é revelar defeitos nas ontologias desenvolvidas.

- Atividade (iv): esta atividade consiste em avaliar os resultados obtidos após a aplicação do teste, com o objetivo de analisar a existência de defeitos e a equivalência entre os mutantes e a ontologia original, assim como calcular o escore de mutação com o intuito de averiguar a confiabilidade do conjunto de dados de teste.

5.1.3 Resultados

A seguir, são apresentados os resultados indivíduais da aplicação do teste de mutação para ontologias OWL em cada uma das ontologias desenvolvidas na atividade (ii) e testadas na atividade (iii) da Figura 5.1. Para cada ontologia são apresentados os operadores de mutação aplicados, o total de mutantes gerados por operador de mutação, o total de mutantes que permaneceram vivos ou foram classificados como inconsistentes após a aplicação dos dados de teste e, o total de mutantes que foram classificados como equivalentes à ontologia original durante a execução da atividade (iv) da Figura 5.1 de avaliação dos resultados.

5.1.3.1 Ontologia Tipos de Esportes - OTE

A Ontologia Tipos de Esportes - OTE, refere-se à uma ontologia com expressividade lógica $\mathcal{ALCIE}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, intersecção e disjunção. Esta ontologia descreve os tipos de esportes coletivos, individuais, local onde ocorrem e a quantidade de pessoas envolvidas em cada esporte. Contém 141 axiomas, sendo 102 axiomas lógicos, 31 classes, 6 propriedades de objetos e 2 propriedades de tipos de dados. Para essa ontologia foi possível a aplicação dos operadores de mutação ACSD, AEDN, CDD, CDU, CUC, ACSTA, PDD, PDU, PDUP, PRU, PDC, PRD e PRUP, conforme os axiomas existentes na ontologia. A Tabela 5.1 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e classificados como equivalentes para cada operador de mutação aplicado.

Tabela 5.1: Resultados do teste de mutação da Ontologia OTE (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACSD	66	14	52	0	0
AEDN	36	0	36	0	0
CDD	2	1	1	0	1
CDU	5	3	2	0	0
CUC	21	3	18	0	0

Continua na próxima página

Tabela 5.1 – *Continua da página anterior*

Operadores	Mutantes				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
EAU	37	37	0	0	0
ACSTA	36	0	36	0	36
PDD	27	3	24	0	3
PDU	13	0	13	0	2
PDUP	10	0	10	0	0
PRU	12	0	12	0	7
PDC	30	12	18	0	18
PRD	27	0	27	0	0
PRUP	9	0	9	0	7
TOTAL	331	73	258	0	74
ESCORE DE MUTAÇÃO	0,22				0,28

Conforme observado na Tabela 5.1 foi obtido um total de 331 mutantes e um escore de mutação de 0,22, dado os 73 mutantes que foram mortos e os 258 que permaneceram vivos após a aplicação do conjunto de 44 dados de teste gerados pelos operadores ACSTA e AEDN.

Através da análise dos mutantes que permaneceram vivos dos operadores ACSD e AEDN foi possível revelar os defeitos de "Intervalo de dados ou domínio limitados" e "Utilizar elementos incorretamente". Ao transformar uma subclasse como disjunta de sua superclasse com o operador ACSD, os resultados da aplicação dos dados de teste não se alteravam devido à limitação do domínio e intervalo de dados definidos na representação das subclasses mutadas. Resultados similares ao operador ACSD foram observados com a aplicação do operador de mutação AEDN. Ao inserir um operando de negação em um axioma lógico, nenhum resultado da aplicação dos dados de teste nos mutantes apresentou diferenças em relação a ontologia original. Dada a ocorrência do defeito de "intervalo de dados ou domínio limitados", ao negar um axioma não causava alteração na ontologia devido a existência desse defeito, fazendo com que o dado de teste aplicado não identificasse a alteração realizada pelo operador de mutação. Esses resultados também possibilitaram a identificação do defeito de "utilizar elementos incorretamente", dado que ao negar um axioma lógico identificou-se a definição incorreta de classes e propriedades de objetos ao invés da definição de propriedades de tipos de dados. A Tabela 5.2 apresenta um exemplo da ocorrência desses dois tipos de defeitos na ontologia.

Tabela 5.2: Ex. do defeito intervalo de dados ou domínio limitado e utilizar elementos incorretamente (autor, 2019)

Classes	**Axioma lógico**
Esportes_Individuais	
Atletismo	temJogador some 1_Jogadores

Conforme a Tabela 5.2, o axioma definido para a subclasse *Atletismo* deveria ser definido de forma mais extensa para a superclasse *Esportes_Individuais*, dado que uma subclasse deve satisfazer as condições de sua superclasse. Por outro lado, o axioma "*temJogador some 1_Jogadores*" deveria ser substituído por um axioma de cardinalidade através de uma propriedade de dados do tipo inteiro, para representar a quantidade de jogadores para um determinado esporte.

Os mutantes que permaneceram vivos dos operadores de mutação CDU e CUC revelaram a existência de um novo tipo de defeito, conforme apresentado na Seção 5.3.1, denominado nesta pesquisa como "Baixo grau de especialização", de modo que os conceitos que representam

os tipos de esportes nesta ontologia, por serem a menor unidade conceitual, deveriam ser representados como indivíduos ao invés de classes. Essa análise também permitiu identificar a existência do defeito de "Definição formal idêntica de classes e instâncias" para essas mesmas classes que representam os tipos de esportes, de modo que a única representação conceitual que diferencia um tipo de esporte de outro é o nome atribuído à propria classe.

Os demais operadores de mutação que apresentaram mutantes vivos também revelaram a existência de um novo tipo de defeito, conforme apresentado na Seção 5.3.1, denominado nesta tese como "Declarar elementos na ontologia e não implementá-los". Este defeito foi revelado pelo fato de terem sido criadas propriedades de objetos nesta ontologia que não foram utilizadas para realizar nenhum tipo de associação entre objetos. Neste caso não foi possível existir um dado de teste capaz de identificar a mutação e matar o mutante.

A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de cinco tipos de defeitos[3], conforme são apresentados a seguir:

- intervalo de dados ou domínio limitados;
- utilizar elementos incorretamente;
- baixo grau de especialização;
- definição formal idêntica de classes e instâncias; e
- declarar elementos na ontologia e não implementá-los.

Considerando que 74 mutantes vivos apresentados na Tabela 5.1 foram classificados como equivalentes, subtraindo estes mutantes do total de mutantes gerados, o escore de mutação para averiguar o grau de adequação dos dados de teste passa a ser 0,28 e não mais 0,22. Este baixo escore de mutação justifica-se pela ocorrência de cinco tipos de defeitos revelados pelos mutantes que permaneceram vivos e não foram classificados como equivalentes, não sendo necessária a geração de um novo conjunto de dados de teste e a continuidade do teste.

5.1.3.2 Ontologia Tipos de Animais - OTA

Similar à ontologia OTE, a Ontologia Tipos de Animais - OTA, também apresenta expressividade lógica $\mathcal{ALCIE}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, intersecção e disjunção. Esta ontologia descreve a classificação dos tipos de animais. Contém 201 axiomas, dos quais 143 são axiomas lógicos, 42 são classes, 13 são propriedades de objetos e 3 são anotações. Para essa ontologia foi possível a aplicação dos operadores de mutação ACSD, AEDN, CDD, CDU, CUC, ACSTA e PDC, conforme os axiomas existentes na ontologia. A Tabela 5.3 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e classificados como equivalentes para cada operador de mutação aplicado.

Tabela 5.3: Resultados do teste de mutação da Ontologia OTA (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACSD	131	121	10	0	0
AEDN	91	82	9	0	0

Continua na próxima página

[3]De acordo com o catálogo de defeitos da Subseção 2.3.2

Tabela 5.3 – *Continua da página anterior*

Operadores	Mutantes				
	Gerados	Mortos	Vivos	Inconsistentes	Equivalentes
CDD	4	4	0	0	0
CDU	1	0	1	0	1
CUC	17	17	0	0	0
EAU	91	91	0	0	0
ACSTA	91	83	8	0	0
PDC	40	23	17	0	8
TOTAL	466	421	45	0	9
ESCORE DE MUTAÇÃO	0,90				0,92

Conforme apresentado na Tabela 5.3 foi obtido um total de 466 mutantes e um escore de mutação de 0,90, dado a baixa quantidade de mutantes que permaneceram vivos após a aplicação do conjunto de 87 dados de teste gerados pelos operadores ACSTA e AEDN.

Pela análise dos mutantes que permaneceram vivos dos operadores ACSD, AEDN e ACSTA, foi possível revelar os defeitos de "Definição formal idêntica de classes e instâncias" e "Assumir características implícitas no nome dos componentes". Ao revelar o defeito de "Definição formal idêntica de classes e instâncias", devido algumas classes terem sido criadas com a mesma representação formal de outras classes, também permitiu identificar o defeito de "Assumir características implícitas no nome dos componentes", de modo que a única representação conceitual que diferencia uma dessas classes de outra é o próprio nome atribuído à elas.

Dos 17 mutantes que permaneceram vivos com o operador PDC, 9 deles também revelaram o defeito de "Definição formal idêntica de classes e instâncias". Os outros 8 mutantes que permaneceram vivos também possibilitaram identificar o defeito de "Assumir características implícitas no nome dos componentes". Porém, em decorrência da existência do defeito de "Definição formal idêntica de classes e instâncias", uma vez corrigido esse defeito, esses 8 operadores também são mortos com os dados de teste. A Figura 5.2 apresenta um exemplo da ocorrência desses dois tipos de defeitos na ontologia.

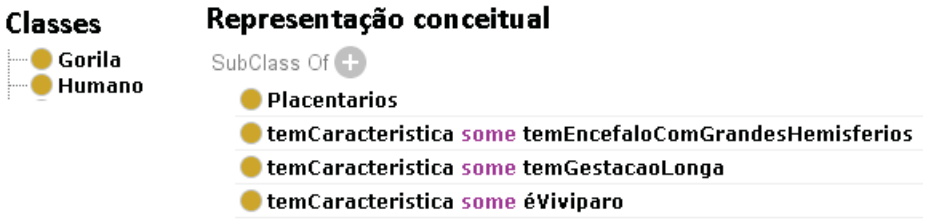


Figura 5.2: Exemplo dos defeitos da ontologia OTA (autor, 2019)

Conforme mostrado na Figura 5.2, as classes *Gorila* e *Humano* são representadas exatamente com os mesmos axiomas lógicos para descrever cada classe, o que impacta que somente o nome de cada classe as diferencia. Porém, como o nome de uma classe não é explícito para o classificador, os defeitos mencionados são revelados.

A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de dois tipos de defeitos[4], conforme seguem:

- Definição formal idêntica de classes e instâncias; e

[4]De acordo com o catálogo de defeitos da Subseção 2.3.2

- assumir características implícitas no nome dos componentes.

Considerando que 9 mutantes vivos apresentados na Tabela 5.3 foram classificados como equivalentes, subtraindo esses mutantes do total de mutantes gerados, o escore de mutação passa a ser 0,92 e não mais 0,90.

5.1.3.3 Ontologia Continentes e Países - OCP

A Ontologia Continentes e Países - OCP, refere-se à uma ontologia com expressividade lógica $\mathcal{ALCHUE}$, consistindo basicamente de axiomas lógico-descritivos de intersecção, união, disjunção, equivalência e restrição existencial. Esta ontologia descreve a classificação dos continentes e seus respectivos países. Contém 153 axiomas, dos quais 107 são axiomas lógicos, 26 são classes, 2 são propriedades de objetos e 18 são indivíduos. Para essa ontologia foi possível a aplicação dos operadores de mutação ACED, ACOTA, ACSD, AEDN, CDD, CDU, CEU, CEUO, CIS, DPC, ACSTA, PRU e PDC, conforme os axiomas existentes na ontologia. A Tabela 5.4 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e classificados como equivalentes à ontologia original para cada operador de mutação aplicado.

Tabela 5.4: Resultados do teste de mutação da Ontologia OCP (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACED	6	6	0	0	0
ACOTA	20	0	14	6	0
ACSD	60	60	0	0	0
AEDN	90	54	0	36	0
CDD	2	2	0	0	0
CDU	18	0	18	0	0
CEU	6	6	0	0	0
CEUO	54	18	36	0	0
CIS	17	17	0	0	0
DPC	6	6	0	0	0
EAU	35	35	0	0	0
UAU	1	1	0	0	0
ACSTA	38	18	2	18	2
PRU	2	2	0	0	0
PDC	24	6	0	18	0
TOTAL	379	229	72	78	6
ESCORE DE MUTAÇÃO	0,76				0,78

Conforme mostrado na Tabela 5.4 foi obtido um total de 379 mutantes e um escore de mutação de 0,76, diante do total de 229 mutantes mortos e dos 78 mutantes que foram classificados como inconsistentes após a aplicação do conjunto de 164 dados de teste gerados pelos operadores ACOTA, AEDN, CEUO e ACSTA.

Ao analisar os mutantes que permaneceram vivos do operador ACOTA revelou-se a existência dos defeitos de "Falta de disjunção" e "Utilização incorreta dos operandos AND e OR". Esses mutantes revelaram esses defeitos na ontologia original devido não haver definições de disjunção para todas as subclasses de uma superclasse. Nos casos em que havia uma definição de

disjunção, essa definição era representada com o operando “OR”, o que não causava efeito entre todas as classes participantes dessa disjunção. Esses defeitos são confirmados com o operador de mutação CDU, onde ao remover as definições de disjunção desta ontologia, todos os mutantes criados permaneceram vivos, indicando que as disjunções definidas nesta ontologia não causam o efeito pretendido. Outra confirmação é observada pelos mutantes inconsistentes gerados com o operador de mutação AEDN. Ao adicionar um operando de negação na definição incorreta da disjunção, é criado um axioma inconsistente que invalida a ontologia. Análise similar é feita para os mutantes que permaneceram vivos do operador de mutação CEUO. Ao remover o operando “OR” dessa definição de disjunção, o mutante apresentava o mesmo resultado da ontologia original com a aplicação dos dados de teste, devido ao uso incorreto do operando “OR”. Na Figura 5.3 é apresentada uma ocorrência desses dois tipos de defeitos revelados.

Figura 5.3: Exemplo dos defeitos da ontologia OCP (autor, 2019)

De acordo com a Figura 5.3, a classe *Brasil* é definida como disjunta somente com as classes *Chile* e *Argentina*, sendo que existem outras classes no mesmo nível hierárquico da ontologia que também deveriam fazer parte dessa disjunção. Por outro lado, a definição de disjunção definida com o operando “OR” entre as classes *Argentina* e *Chile*, possibilita que a classe *Brasil* seja equivalente à uma delas e disjunta de outra.

Os mutantes inconsistentes produzidos pelo operador de mutação ACSTA referem-se às ontologias estarem relacionadas a condição de Suposição de Mundo Aberto - OWA (do inglês, *Open World Assumption*). Como existe mais de um axioma existencial para uma determinada classe, ao modificar somente um desses axiomas para universal e existir uma definição de disjunção entre as classes representadas nesses axiomas, o novo axioma mutado produz uma insconsistência na ontologia, dado que deveria ser criado um axioma universal que abrangesse todos os axiomas existenciais dessa classe, impendindo que novos conceitos sejam adicionados. A Figura 5.4 apresenta um exemplo dessa inconsistência.

Figura 5.4: Exemplo de mutante inconsistente do operador ACSTA (autor, 2019)

Conforme mostrado na Figura 5.4, ao mutar o axioma “*temPais some Argentina*” para “*temPais only Argentina*” e como existe uma definição de disjunção entre as classes *Brasil*, *Chile* e *Argentina* (Figura 5.3), a ontologia torna-se inconsistente por não ser possível satistazer esta condição. Processo similar ocorre com os mutantes inconsistentes gerados pelo operador de mutação PDC. Porém, a inconsistência gerada pela mutação do operador PDC permitiu revelar o

defeito de “Redundância gramatical” e “Definição formal idêntica de classes e instâncias”. Ao transformar uma classe primitiva de um país em uma classe definida, uma inconsistência era gerada devido a classe que representa o continente desse país possuir a mesma representação conceitual, ocasionando uma redundância na ontologia e tornando-a inconsistente devido a disjunção entre as classes de países. A Figura 5.5 apresenta um exemplo dessa inconsistência.

Figura 5.5: Exemplo de mutante inconsistente do operador PDC (autor, 2019)

De acordo com a Figura 5.5, ao transformar a classe *Argentina* de subclasse para equivalente, gera um conflito com as definições da classe *America_do_Sul*, devido a definição equívocada de disjunção entre as classes *Argentina*, *Brasil* e *Chile* (Figura 5.3) não permitir que essas classes tenham indivíduos em comum entre si e com a classe *America_do_Sul*.

A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de quatro tipos de defeitos[5], conforme são apresentados a seguir:

- falta de disjunção;
- utilização incorreta dos operandos AND e OR;
- redundância gramatical; e
- definição formal idêntica de classes e instâncias.

Considerando que 6 mutantes vivos apresentados na Tabela 5.3 foram classificados como equivalentes, subtraindo esses mutantes do total de mutantes gerados, o escore de mutação passa a ser 0,78 e não mais 0,76.

5.1.3.4 Ontologia Classificação de Esportes - OCE

A Ontologia Classificação de Esportes - OCE, similar à ontologia OCP, também refere-se à uma ontologia com expressividade lógica $\mathcal{ALCHUE}$, consistindo basicamente de axiomas lógico-descritivos de intersecção, união, disjunção, equivalência e restrição existencial. Esta ontologia descreve os tipos de esportes, onde são praticados e materiais utilizados em cada modalidade. Contém 170 axiomas, dos quais 116 são axiomas lógicos, 51 são classes e 3 são propriedades de objetos. Para essa ontologia foi possível a aplicação dos operadores de mutação ACED, ACOTA, ACSD, AEDN, AEUN, CDD, CDU, CEU, CEUO, CUC, DPC, ACSTA, PDD, PDU, PRU, PDC e PRD, conforme os axiomas existentes na ontologia. A Tabela 5.5 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e classificados como equivalentes à ontologia original para cada operador de mutação aplicado.

[5]De acordo com o catálogo de defeitos da Subseção 2.3.2

Tabela 5.5: Resultados do teste de mutação da Ontologia OCE (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACED	1	1	0	0	0
ACOTA	4	4	0	0	0
ACSD	107	107	0	0	0
AEDN	68	68	0	0	0
AEUN	13	13	0	0	0
CDD	6	6	0	0	0
CDU	3	3	0	0	0
CEU	1	1	0	0	0
CEUO	10	10	0	0	0
CUC	18	18	0	0	0
DPC	1	0	1	0	1
EAU	59	59	0	0	0
ACSTA	59	59	0	0	0
PDD	12	12	0	0	0
PDU	2	2	0	0	0
PRU	2	2	0	0	0
PDC	39	39	0	0	0
PRD	8	8	0	0	0
TOTAL	413	412	1	0	1
ESCORE DE MUTAÇÃO	0,99				1

Conforme mostrado na Tabela 5.5 foi obtido um total de 413 mutantes e um escore de mutação de 0,99, dado que somente um mutante permaneceu vivo após a aplicação do conjunto de 127 dados de teste gerados pelos operadores ACOTA, AEDN, AEUN, CEUO e ACSTA. Como o único mutante que permaneceu vivo foi considerado equivalente à ontologia original, nenhum defeito foi revelado. Desse modo, o escore de mutação com índice de 0,99 é considerado como 1, demonstrando a qualidade dos dados de teste utilizados.

5.1.3.5 Ontologia Times de Futebol Americano - OTFA

A Ontologia Times de Futebol Americano - OTFA, apresenta expressividade lógica $\mathcal{ALCE}$, consistindo basicamente de axiomas lógico-descritivos de intersecção, disjunção e restrição existencial. Esta ontologia descreve os times da liga de futebol americano dos Estados Unidos e as cidades que eles representam. Contém 298 axiomas, sendo 187 axiomas lógicos, 76 classes, 2 propriedades de objetos, 1 propriedade de tipos de dados e 32 indivíduos. Para essa ontologia foi possível a aplicação dos operadores de mutação ACSD, AEDN, CDU, CIS, CUC, ACSTA, PDD, PDU, PRU, PDC e PRD. A Tabela 5.6 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes para cada operador de mutação aplicado.

Tabela 5.6: Resultados do teste de mutação da Ontologia OTFA (autor, 2019)

Operadores	Mutantes				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACSD	138	72	0	66	0
AEDN	63	63	0	0	0
CDU	12	12	0	0	0
CIS	32	32	0	0	0
CUC	24	24	0	0	0
EAU	64	64	0	0	0
ACSTA	64	62	0	2	0
PDD	32	32	0	0	0
PDU	2	0	2	0	2
PRU	2	0	2	0	2
PDC	74	64	0	10	0
PRD	32	0	1	31	1
TOTAL	539	425	5	109	5
ESCORE DE MUTAÇÃO	0,98				1

Conforme mostrado na Tabela 5.6, foi obtido um total de 539 mutantes e um escore de mutação de 0,98 ao desconsiderar os mutantes inconsistentes do total de mutantes gerados, após a aplicação do conjunto de 300 dados de teste produzidos pelos operadores AEDN e ACSTA. Nenhum defeito foi revelado nesta ontologia, dado que os 5 mutantes que permaneceram vivos foram considerados equivalentes à ontologia original. Esta ontologia gerou um número consideravelmente alto de mutantes inconsistentes, porém nenhum deles apresentou relação com algum tipo de defeito. A inconsistência desses mutantes foi causada pelo motivo de os axiomas estarem representados de forma correta e ao serem mutados entrarem em conflito com os axiomas de disjunção da ontologia. Deste modo, como todos os mutantes vivos foram classificados como equivalentes à ontologia original, obtém-se um escore de mutação com índice 1, o que demonstra a qualidade do conjunto de dados de teste utilizado.

5.1.3.6 Ontologia Pontos Turísticos - OPT

A Ontologia Pontos Turísticos - OPT, similar à ontologia OTFA, também apresenta expressividade lógica $\mathcal{ALCE}$, consistindo basicamente de axiomas lógico-descritivos de intersecção, disjunção e restrição existencial. Esta ontologia descreve os pontos turísticos e as cidades e países onde estes se localizam. Contém 586 axiomas, sendo 421 axiomas lógicos, 159 classes e 6 propriedades de objetos. Para essa ontologia foi possível a aplicação dos operadores de mutação ACSD, AEDN, CDD, CDU, CUC, ACSTA, PDD, PDU, PDUP, PRU, PDC, PRD e PRUP. A Tabela 5.7 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes a ontologia original para cada operador de mutação aplicado.

Tabela 5.7: Resultados do teste de mutação da Ontologia OPT (autor, 2019)

Operadores	Mutantes				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACSD	403	403	0	0	0

Continua na próxima página

Tabela 5.7 – *Continua da página anterior*

Operadores	Mutantes				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
AEDN	246	246	0	0	0
CDD	4	4	0	0	0
CDU	2	2	0	0	0
CUC	54	0	54	0	54
EAU	247	247	0	0	0
ACSTA	246	246	0	0	0
PDD	238	238	0	0	0
PDU	6	6	0	0	0
PDUP	5	0	5	0	5
PRU	6	0	6	0	6
PDC	157	152	5	0	0
PRD	6	0	6	0	6
PRUP	6	0	6	0	6
TOTAL	1626	1544	82	0	77
ESCORE DE MUTAÇÃO	0,94				0,99

Conforme mostrado na Tabela 5.7 foi obtido um total de 1626 mutantes e um escore de mutação de 0,94, devido a baixa quantidade de mutantes que permaneceram vivos após a aplicação do conjunto de 952 dados de teste gerados pelos operadores AEDN e ACSTA e por não ter sido gerado nenhum mutante inconsistente. Todos os mutantes gerados pelo operador de mutação CUC permaneceram vivos e foram considerados equivalentes à ontologia original. Isto ocorreu devido a correta definição do domínio e intervalo das propriedades utilizadas na representação conceitual das classes que sofreram as mutações. Deste modo, ao alterar a superclasse de uma subclasse, o classificador automaticamente inferia que essa subclasse continuava pertencendo a sua antiga superclasse. Os mutantes vivos do operador PDC revelaram o defeito de "Utilizar diferentes critérios de nomenclatura". Esse defeito foi identificado porque as classes utilizadas nas expressões lógicas desses mutantes foram nomeadas utilizando o caractere apóstrofo (') na sua descrição, o que impossibilita a execução dos dados de teste que continham o nome dessas classes. A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de um tipo de defeito[6], conforme é apresentado a seguir:

- utilizar diferentes critérios de nomenclatura.

Considerando que 77 mutantes vivos apresentados na Tabela 5.7 foram classificados como equivalentes, o escore de mutação passa a ser de 0,99 e não mais de 0,94.

5.1.4 Análise dos resultados do Experimento I

Para cada ontologia desenvolvida na atividade (ii) (Subseção 5.1.2), inicialmente foi proposto aplicar todos os operadores de mutação (Seção 3.4) durante os testes da atividade (iii). Porém, foram utilizados operadores de mutação compatíveis com os tipos de axiomas definidos em cada uma das ontologias originais. Deste modo, os operadores utilizados neste experimento não são necessariamente os mesmos para todas as ontologias, variando de acordo com os axiomas

[6]De acordo com o catálogo de defeitos da Subseção 2.3.2

definidos em cada uma delas. Os operadores de mutação CEUA, ACATO, ACATS, ACMiMa, ACMaMi, ACMiEx, ACMaEx, ACEMi, ACEMa, MaCAU, MiCAU, ECAU, ACTF, ACFT, IDU e ISU, não foram utilizados neste experimento por não existirem axiomas nas ontologias que estivessem de acordo com o tipo de mutação realizada por estes operadores. Isto não inviabiliza o método de teste, pelo fato de que mesmo os operadores terem sido desenvolvidos para revelar determinados tipos de defeitos, cada defeito pode ser revelado por mais de um operador e, cada operador pode também revelar outros tipos de defeitos para o qual ele foi desenvolvido. Assim, não é necessária a aplicação de todos os operadores de mutação para revelar um determinado tipo defeito na ontologia. Nas 6 ontologias testadas no Experimento I, foram utilizados 14 operadores de mutação estrutural e 8 operadores lógicos de mutação, conforme é apresentado na Tabela 5.8.

Tabela 5.8: Operadores de mutação utilizados em cada ontologia (autor, 2019)

Operadores Estruturais	**Ontologias**					
	OTE	**OTA**	**OCP**	**OCE**	**OTFA**	**OPT**
ACED			X	X		
ACSD	X	X	X	X	X	X
CDD	X	X	X	X		X
CDU	X	X	X	X	X	X
CEU			X	X		
CUC		X		X	X	X
PDD	X			X	X	X
PDU	X			X	X	X
PDUP	X					X
PRU	X		X	X	X	X
DPC			X	X		
PDC	X	X	X	X	X	X
PRD	X			X	X	X
PRUP	X					X
Operadores Lógicos	**Ontologias**					
	OTE	**OTA**	**OCP**	**OCE**	**OTFA**	**OPT**
ACOTA			X	X		
AEDN	X	X	X	X	X	X
AEUN				X		
CEUO			X	X		
CIS			X		X	
EAU	X	X	X	X	X	X
UAU			X			
ACSTA	X	X	X	X	X	X

Somente os operadores de mutação estrutural ACSD, CDU e PDC e os operadores lógicos de mutação AEDN, EAU e ACSTA foram utilizados em todas as ontologias deste experimento. É possível observar a alta ocorrência do uso dos elementos mutados por esses operadores no desenvolvimento de ontologias e também a incidência de defeitos na implementação desses elementos, visto que desses 6 operadores de mutação, 5 operadores revelaram defeitos nas ontologias, conforme pode ser observado na Tabela 5.9. Os demais operadores da Tabela 5.8 foram aplicados conforme a existência de elementos compatíveis com as mutações realizadas por

cada operador de mutação, sendo que dos 22 operadores aplicados 14 revelaram pelo menos um tipo de defeito.

Tabela 5.9: Operadores de mutação que revelaram defeitos por ontologia (autor, 2019)

Operadores Estruturais	**Ontologias**					
	OTE	**OTA**	**OCP**	**OCE**	**OTFA**	**OPT**
ACSD	X	X				
CDU	X		X			X
CUC	X					
PDD	X					
PDU	X					
PDUP	X					
PRU	X					
PDC	X	X	X	X	X	X
PRD	X	X				
PRUP	X					X
Operadores Lógicos	**Ontologias**					
	OTE	**OTA**	**OCP**	**OCE**	**OTFA**	**OPT**
ACOTA			X			
ACSTA		X				
AEDN	X					
CEUO			X			

Os operadores ACSD, AEDN, PRD, PRUP, PDC e ACSTA produziram mutantes inconsistentes que não permitem avaliar os dados de teste, porém, todos esses operadores revelaram defeitos nas ontologias, e a inconsistência gerada pelo operador AEDN auxiliou na revelação dos defeitos da ontologia OCP.

Os operadores de mutação estrutural ACED, CDD, CEU e DPC e os operadores lógicos de mutação AEUN, EUA, UAU e CIS apresentaram todos os seus mutantes mortos pelos dados de teste, de modo que não revelaram nenhum tipo de defeito e também não apresentaram mutantes equivalentes à ontologia original. O operador EUA foi aplicado em todas as ontologias testadas, o operador CDD foi aplicado em 5 das 6 ontologias, já os demais operadores, por não existirem elementos compátiveis com as mutações, foram aplicados em 2 das 5 ontologias, exceto para o operador AEUN que foi aplicado somente na ontologia OCE. Não é possível afirmar que esses operadores não revelariam algum tipo de defeito nas demais ontologias, caso tivessem elementos compatíveis com as mutações. Porém, esses operadores possibilitaram avaliar a qualidade dos dados de teste, de modo que os dados aplicados identificaram todos os defeitos simulados por estes operadores. A Tabela 5.10 apresenta os operadores de mutação que não revelaram defeitos e em quais ontologias foram aplicados.

Tabela 5.10: Operadores de mutação que não revelaram defeitos (autor, 2019)

Operadores Estruturais	**Ontologias**					
	OTE	**OTA**	**OCP**	**OCE**	**OTFA**	**OPT**
ACED			X	X		
CDD	X	X	X	X		X
CEU			X	X		

Continua na próxima página

Tabela 5.10 – *Continua da página anterior*

DPC			X	X		
Operadores	**Ontologias**					
Lógicos	**OTE**	**OTA**	**OCP**	**OCE**	**OTFA**	**OPT**
AEUN				X		
EUA	X	X	X	X	X	X
UAU			X			
CIS			X		X	

Ao todo foram identificados 10 tipos de defeitos cometidos pelos desenvolvedores nas ontologias testadas, dos quais 8 fazem parte da lista de defeitos apresentada na Subseção 2.3.2, sendo os defeitos intervalo de dados ou domínio limitados, utilizar elementos incorretamente, definição formal idêntica de classes e instâncias, assumir características implícitas no nome dos componentes, falta de disjunção, utilização incorreta dos operandos AND e OR, redundância gramatical e utilizar diferentes critérios de nomenclatura. Os defeitos baixo grau de especialização e declarar elementos na ontologia e não implementá-los, identificados na ontologia OTE, não são apresentados na lista de defeitos da Subseção 2.3.2. Estes defeitos foram classificados como novos tipos de defeitos identificados pelos operadores de mutação CDU e CUC para o defeito de baixo grau de especialização e, pelos operadores PDD, PDU, PDUP, PRU, PRD e PRUP para o defeito de definir elementos na ontologia e não utilizá-los.

Os conjuntos de dados de teste gerados para cada uma das ontologias apresentaram resultados satisfatórios diante do escore de mutação obtido, sendo que após a definição dos mutantes equivalentes os conjuntos de dados de teste das ontologias OCE e OTFA apresentaram escores de mutação de 100% e, os conjuntos de dados de teste das ontologias OPT e OTA de 99% e 92% respectivamente. O conjunto de dados de teste da ontologia OCP apresentou o escore de mutação de 77%, devido ao total de 78 mutantes inconsistentes que foram gerados e não foram considerados como mortos pelos dados de teste no cálculo do escore de mutação, e portanto não desqualificam os dados de teste gerados. Já o conjunto de dados de teste gerados para a ontologia OTE apresentou um escore de mutação de 28%, devido ao total de 258 mutantes que permaneceram vivos e permitiram revelar 5 tipos de defeitos nessa ontologia.

5.2 EXPERIMENTO II

Neste experimento são consideradas ontologias da área biomédica, selecionadas a partir da base de ontologias disponilizadas pelo grupo *OBO Foundry*[7]. Este grupo trata-se de um experimento colaborativo que envolve desenvolvedores de ontologias que utilizam um conjunto crescente de princípios que especificam as melhores práticas no desenvolvimento de ontologias. Ao todo foram selecionadas 6 ontologias, sendo a mesma quantidade utilizada no Experimento I.

5.2.1 Motivação

No Experimento I foram utilizadas ontologias OWL construídas por desenvolvedores com pouca experiência e sem considerar o uso de métodos ou práticas para o desenvolvimento e avaliação de ontologias, o que possibilita que um número maior de defeitos sejam cometidos. No Experimento II, são consideradas ontologias OWL desenvolvidas por especialistas que utilizam princípios que especificam as melhores práticas no desenvolvimento, indicando que um número menor de defeitos ou até mesmo nenhum seja cometido pelo desenvolvedor. Deste modo,

[7]http://www.obofoundry.org/

pretende-se com este experimento analisar além das questões de pesquisa do Experimento I, a seguinte questão:

5. É possível revelar defeitos com o teste de mutação para ontologias OWL proposto nesta tese em ontologias construídas por especialistas do domínio e especialistas no desenvolvimento de ontologias?

- **Objetivo**
  - Avaliar o método de teste de mutação para ontologias OWL em ontologias construídas por especialistas do domínio e especialistas no desenvolvimento de ontologias, para avaliar a identificação de defeitos e adequação dos dados de teste.

5.2.2 Método

O Experimento II é baseado em três atividades distintas: (i) seleção de 6 ontologias da base de ontologias biomédicas *OBO Foundry*, (ii) cada ontologia foi testada utilizando o método do teste de mutação para ontologias proposto nesta tese. A última atividade consiste em (iii) avaliar os resultados obtidos após a aplicação do teste, com relação ao número de mutantes mortos e vivos, o que possibilita avaliar a existência de defeitos e a equivalência entre os mutantes e a ontologia original. A Figura 5.6 apresenta a sequência das atividades.

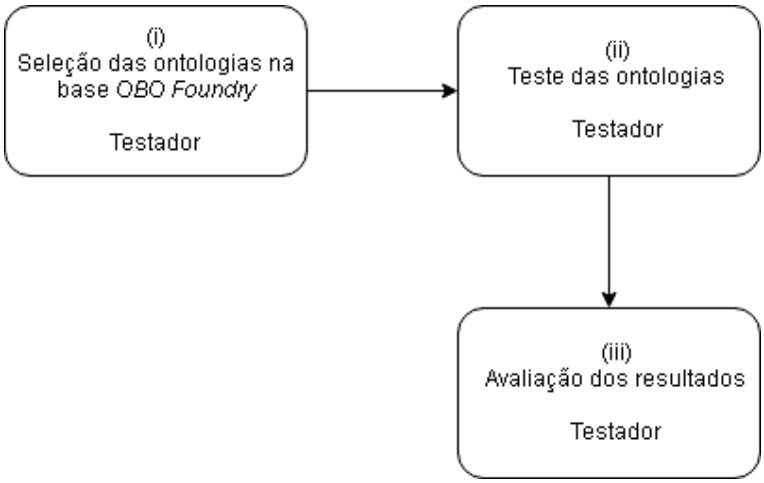


Figura 5.6: Metodologia do Experimento II (autor, 2019)

Cada atividade apresentada na Figura 5.6 possui os seguintes objetivos:

- Atividade (i): esta atividade tem como objetivo a seleção de um conjunto de ontologias desenvolvidas por desenvolvedores especialistas em ontologias e no domínio do conhecimento ao qual a ontologia aborda. Esta atividade justifica-se devido no Experimento I ser utilizado um conjunto de ontologias desenvolvidas por desenvolvedores não especialistas, o que possibilita avaliar o método do teste de mutação para ontologias OWL em dois cenários distintos.

- Atividade (ii): esta atividade tem o mesmo objetivo da atividade (iii) do Experimento I (Figura 5.1), que é validar o conjunto de operadores de mutação, assim como a ferramenta para geração dos mutantes e dados de teste e o método de teste de mutação para ontologias OWL proposto nesta tese.

- Atividade (iii): assim como na atividade (iv) do Experimento I (FIgura 5.1), esta atividade consiste em avaliar os resultados obtidos após a aplicação do teste, com o objetivo de analisar a existência de defeitos e a equivalência entre os mutantes e a ontologia original, assim como calcular o escore de mutação com o intuito de averiguar a confiabilidade do conjunto de dados de teste gerado com os operadores de mutação.

5.2.3 Resultados

A seguir, são apresentados os resultados indivíduais da aplicação do teste de mutação para ontologias OWL em cada uma das ontologias selecionadas na atividade (i) da Figura 5.6 e testadas na atividade (ii) da mesma figura. Para cada ontologia são apresentados os operadores de mutação aplicados, o total de mutantes gerados por operador de mutação, o total de mutantes que permaneceram vivos ou foram classificados como inconsistentes após a aplicação dos dados de teste e, o total de mutantes que foram classificados como equivalentes à ontologia original durante a execução da atividade (iii) de avaliação dos resultados da Figura 5.6.

5.2.3.1 Ontologia Comparativa de Análise de Dados - CDAO

A Ontologia Comparativa de Análise de Dados - CDAO (do inglês, *Comparative Data Analysis Ontology*)[8], fornece uma estrutura para a compreensão de dados no contexto da análise comparativa evolutiva na área da biologia. Esta ontologia é desenvolvida por cientistas em biologia, evolução e ciência da computação. Possui expressividade lógica $\mathcal{ALCUNHE}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, universal, intersecção, união, cardinalidades, equivalências e disjunção. Contém 649 axiomas, sendo 421 axiomas lógicos, 132 classes, 69 propriedades de objetos, 9 propriedades de tipos de dados, 8 indivíduos e 2 propriedades de anotações. A Tabela 5.11 apresenta os operadores de mutação aplicados nessa ontologia, o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes para cada operador de mutação.

Tabela 5.11: Resultados do teste de mutação da Ontologia CDAO (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACATO	8	8	0	0	0
ACOTA	18	18	0	0	0
ACED	10	10	0	0	0
ACSD	180	180	0	0	0
AEDN	92	92	0	0	0
AEUN	1	1	0	0	0
CDD	31	30	1	0	1
CEU	11	11	0	0	0
CDU	7	3	4	0	4
CEUA	18	12	6	0	6
CEUO	9	9	0	0	0
CIS	16	16	0	0	0
CUC	70	64	6	0	6
ACEMa	15	12	3	0	3

Continua na próxima página

[8]http://www.obofoundry.org/ontology/cdao.html

Tabela 5.11 – *Continua da página anterior*

Operadores	Mutantes				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACEMi	15	6	9	0	9
ACATS	14	14	0	0	0
ACSTA	26	26	0	0	0
ECAU	13	13	0	0	0
EAU	24	24	0	0	0
UAU	14	14	0	0	0
DPC	11	10	1	0	1
PDC	127	117	10	0	10
PDD	83	81	2	0	2
PDU	59	7	52	0	2
PDUP	17	3	14	0	0
PRU	59	29	30	0	0
PRD	130	63	67	0	0
PRUP	16	6	10	0	0
ACMaEx	1	1	0	0	0
ACMaMi	1	1	0	0	0
ACMiEx	6	6	0	0	0
ACMiMa	7	7	0	0	0
MaCAU	1	1	0	0	0
MiCAU	1	1	0	0	0
TOTAL	1111	896	215	0	44
ESCORE DE MUTAÇÃO	0,81				0,84

De acordo com a Tabela 5.11 foi obtido um total de 1111 mutantes e um escore de mutação de 0,81, devido aos 215 mutantes que permaneceram vivos após a aplicação do conjunto de 356 dados de teste gerados pelos operadores ACATO, ACOTA, AEDN, AEUN, CEUA, CEUO, ACEMa, ACEMi, ACATS, ACSTA, ECAU, ACMaEx, ACMiEx, ACMaMi, MaCAU e MiCAU.

A análise dos mutantes que permaneceram vivos dos operadores PDUP, PRD e PRUP permitiu revelar o novo tipo de defeito identificado no experimento I e que é apresentado na Seção 5.3.1, denominado nesta pesquisa como “declarar elementos na ontologia e não implementá-los”. Como esses mutantes referem-se a alterações realizadas no domínio e intervalo tanto de propriedades de objetos como propriedades de tipos de dados, essas propriedades foram criadas na ontologia mas não foram utilizadas na representação de nenhum conceito. Deste modo, não causa diferença alguma na ontologia qualquer tipo de mutação em um elemento que não esteja sendo usado para representar qualquer conceito. Já para os mutantes que permaneceram vivos dos operadores de mutação PDU e PRU, além do defeito de “declarar elementos na ontologia e não implementá-los”, também foi identificado o defeito de “intervalo de dados ou domínio limitado”. Do total de 52 mutantes vivos do operador PDU, 32 revelaram o tipo de defeito “declarar elementos na ontologia e não implementá-los” e 18 mutantes revelaram o tipo de defeito “intervalo de dados ou domínio limitado”. O mesmo ocorreu para os mutantes vivos do operador de mutação PRU, onde do total de 30 mutantes vivos, 27 revelaram o defeito “declarar elementos na ontologia e não implementá-los” e 3 mutantes revelaram o defeito “intervalo de dados ou domínio limitado”.

A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de dois tipos de defeitos[9], conforme são apresentados a seguir:

- intervalo de dados ou domínio limitados; e
- declarar elementos na ontologia e não implementá-los.

Considerando que 44 mutantes apresentados na Tabela 5.11 foram classificados como equivalentes à ontologia original, é possível considerar o escore de mutação para averiguar o grau de adequação dos dados de teste utilizados nesta ontologia como 0,84 ao invés de 0,81.

5.2.3.2 Ontologia Doenças Infecciosas - IDO

A Ontologia Doenças Infecciosas - IDO (do inglês, *Infectious Disease Ontology*)[10], é um conjunto de ontologias interoperáveis que juntas fornecem a cobertura do domínio de doenças infecciosas. O núcleo desse conjunto de ontologias é composto por uma ontologia de nível superior, a qual foi testada neste experimento, que hospeda termos de relevância geral em todo o domínio. É uma ontologia que apresenta expressividade lógica $\mathcal{ALCHUE}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, universal, intersecção, união, equivalências e disjunção. Contém 1730 axiomas, sendo 1087 axiomas lógicos, 519 classes, 38 propriedades de objetos, 20 indivíduos e 66 propriedades de anotações. Para essa ontologia foi possível a aplicação dos operadores de mutação ACATO, ACOTA, ACED, ACSD, AEDN, AEUN, CDD, CDU, CEU, CEUA, CEUO, CIS, CUC, DPC, PDC, IDU, ACATS, ACSTA, PDD, PDU, PDUP, PRU, PRD e PRUP, conforme os axiomas existentes na ontologia. A Tabela 5.12 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes à ontologia original para cada operador de mutação aplicado.

Tabela 5.12: Resultados do teste de mutação da Ontologia IDO (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACATO	208	189	0	19	0
ACOTA	12	12	0	0	0
ACED	84	80	0	4	0
ACSD	598	597	0	1	0
AEDN	535	535	0	0	0
AEUN	4	4	0	0	0
CDD	48	48	0	0	0
CEU	84	84	0	0	0
CDU	312	312	0	0	0
CEUA	311	271	38	2	16
CEUO	33	33	0	0	0
CIS	18	18	0	0	0
CUC	453	453	0	0	0
EAU	104	104	0	0	0
UAU	38	38	0	0	0

Continua na próxima página

[9]De acordo com o catálogo de defeitos da Subseção 2.3.2

[10]http://www.obofoundry.org/ontology/ido.html

Tabela 5.12 – *Continua da página anterior*

Operadores	Mutantes				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACATS	49	49	0	0	0
ACSTA	266	260	4	2	4
DPC	84	81	3	0	3
PDC	518	518	0	0	0
PDD	89	33	56	0	0
PDU	16	9	7	0	0
PDUP	16	9	7	0	0
PRU	18	8	10	0	0
PRD	153	98	55	0	0
PRUP	18	8	10	0	0
IDU	2	0	2	0	0
TOTAL	4071	3851	192	28	23
ESCORE DE MUTAÇÃO	0,95				0,96

Conforme mostrado na Tabela 5.12, foi obtido um total de 4071 mutantes e um escore de mutação de 0,95, dado a baixa quantidade de mutantes que permaneceram vivos e somente 28 mutantes inconsistentes após a aplicação do conjunto de 1095 dados de teste gerados pelos operadores ACATO, ACOTA, AEDN, AEUN, CEUA, CEUO, ACATS e ACSTA.

O operador de mutação ACATO apresentou 19 mutantes inconsistentes ao trocar o operando “AND” pelo operando “OR” em um axioma de equivalência, devido as classes representadas por esses axiomas possuírem definições de disjunção que invalidavam o axioma mutado. Para os 38 mutantes que permaneceram vivos do operador de mutação CEUA, 1 apresentou o defeito de “uso equivocado de qualificadores de restrição”, 9 apresentaram o defeito de “falta de domínio ou intervalo nas propriedades”, 12 revelaram o defeito de “assumir características implícitas no nome dos componentes” e 16 foram classificados como equivalentes. Os 2 mutantes vivos do operador IDU também revelaram o defeito de “assumir características implícitas no nome dos componentes”, devido ao fato de não existir nenhum axioma descritivo para esses indivíduos que justificasse o axioma de disjunção ou a sua ausência.

Na análise dos mutantes que permaneceram vivos dos operadores de mutação PDD, PDU, PDUP, PRU, PRD e PRUP, foi revelado o novo defeito denominado “declarar elementos na ontologia e não implementá-los”. Como todos esses operadores aplicam mutações em axiomas para propriedades tanto de objetos como de tipos de dados, as propriedades que sofreram essas mutações foram definidas na ontologia mas não foram utilizadas na representação de nenhum conceito. A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de quatro tipos de defeitos[11], conforme são apresentados a seguir:

- uso equivocado de qualificadores de restrição;
- falta de domínio ou intervalo nas propriedades;
- assumir características implícitas no nome dos componentes; e
- declarar elementos na ontologia e não implementá-los.

[11]De acordo com o catálogo de defeitos da Subseção 2.3.2

Como 23 mutantes foram classificados como equivalentes à ontologia original, ao subtraí-los do total de mutantes gerados é obtido o escore de mutação de 0,96 ao invés de 0,95.

5.2.3.3 Ontologia Obstétrica e Neonatal - ONTONEO

A Ontologia Obstétrica e Neonatal - ONTONEO (do inglês, *Obstetric and Neonatal Ontology*)[12], é um vocabulário controlado estruturado para fornecer uma representação dos dados dos registros eletrônicos de saúde - EHR (do inglês, *Eletronic Health Record*) envolvidos no cuidado de gestantes e bebês. Esta ontologia possui expressividade lógica $\mathcal{ALCHUEN}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, universal, intersecção, união, cardinalidades e equivalências. Contém 782 axiomas, dos quais 430 são axiomas lógicos, 291 são classes, 46 são propriedades de objetos, 1 é propriedade de tipos de dados, 3 são indivíduos e 11 são propriedades de anotações. Para essa ontologia foi possível a aplicação dos operadores de mutação ACATO, ACOTA, ACED, ACSD, AEDN, AEUN, CDD, CDU, CEU, CEUA, CEUO, CIS, CUC, ACMiEx, ACMiMa, MiCAU, ACEMa, ACEMi, ACATS, ACSTA, ECAU, DPC, PDC, PDD, PDU, PRU e PRD, conforme os axiomas existentes na ontologia. A Tabela 5.13 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes à ontologia original para cada operador de mutação aplicado.

Tabela 5.13: Resultados do teste de mutação da Ontologia ONTONEO (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACATO	104	104	2	0	2
ACOTA	9	7	2	0	2
ACED	21	21	0	0	0
ACSD	389	389	0	0	0
AEDN	516	516	0	0	0
AEUN	1	1	0	0	0
CDD	68	68	0	0	0
CEU	21	21	0	0	0
CDU	1	1	0	0	0
CEUA	256	237	19	0	10
CEUO	22	22	0	0	0
CIS	3	3	0	0	0
CUC	119	119	0	0	0
EAU	73	73	0	0	0
UAU	2	2	0	0	0
ACATS	2	2	0	0	0
ACSTA	248	248	0	0	0
ACMiEx	2	2	0	0	0
ACMiMa	2	2	0	0	0
MiCAU	1	1	0	0	0
ACEMa	3	1	2	0	0
ACEMi	3	1	2	0	0
ECAU	1	1	0	0	0

Continua na próxima página

[12]http://www.obofoundry.org/ontology/ontoneo.html

Tabela 5.13 – *Continua da página anterior*

Operadores	Mutantes				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
DPC	21	21	0	0	0
PDC	364	364	0	0	0
PDD	9	6	3	0	1
PDU	6	4	2	0	1
PRU	7	6	1	0	0
PRD	9	8	0	0	0
TOTAL	2285	2250	35	0	16
ESCORE DE MUTAÇÃO	0,98				0,99

De acordo com a Tabela 5.13 foi obtido um total de 2285 mutantes e um escore de mutação de 0,98, dado que dos 35 mutantes que permaneceram vivos, 16 foram classificados como equivalentes após a aplicação do conjunto de 1443 dados de teste gerados pelos operadores ACATO, ACOTA, AEDN, AEUN, CEUA, CEUO, ACATS, ACSTA, ACMiEx, ACMiMa, MiCAU, ACEMa, ACEMi e ECAU.

Os mutantes que permaneceram vivos dos operadores de mutação ACEMa, ACEMi e ECAU revelaram o defeito de "redundância gramatical", devido estes mutantes apresentarem o axioma que sofreu a mutação repetido duas vezes para representar um conceito na ontologia. Os mutantes vivos do operador ACEMi também revelaram o defeito "*MinIsZero*", dado que além do axioma ser redundante, possuía cardinalidade mínima definida como 0 (zero). Os mutantes vivos dos operadores PDD, PDU, PRU e PRD revelaram o defeiro "declarar elementos na ontologia e não implementá-los". Conforme ocorreu nas outras ontologias testadas, algumas propriedades foram definidas e não foram utilizadas na representação de nenhum conceito da ontologia. A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de três tipos de defeitos[13], conforme são apresentados a seguir:

- redundância gramatical;
- *minIsZero*; e
- declarar elementos na ontologia e não implementá-los.

Ao subtrair o total de mutantes equivalentes do total de mutantes gerados é obtido o escore de mutação para averiguar o grau de adequação dos dados de teste utilizados nesta ontologia como 0,99 ao invés de 0,98.

5.2.3.4 Ontologia Estruturas Organizacionais de Centros e Sistemas de Trauma - OOSTT

A Ontologia Estruturas Organizacionais de Centros de Trauma e Sistemas de Trauma - OOSTT (do inglês, *Ontology of Organizational Structures of Trauma centers and Trauma systems*)[14], é uma ontologia construída para representar os componentes organizacionais de centros e sistemas de trauma. Esta ontologia possui expressividade lógica $\mathcal{ALCHUE}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, universal, intersecção, união, equivalências e negação. Contém 940 axiomas, sendo 551 axiomas lógicos, 347 classes, 34

[13]De acordo com o catálogo de defeitos da Subseção 2.3.2

[14]http://www.obofoundry.org/ontology/oostt.html

propriedades de objetos, 1 propriedade de tipos de dados, 2 indivíduos e 5 propriedades de anotações. A Tabela 5.14 apresenta os operadores de mutação aplicados nessa ontologia, o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes à ontologia original para cada operador de mutação.

Tabela 5.14: Resultados do teste de mutação da Ontologia OOSTT (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACATO	183	183	0	0	0
ACOTA	5	5	0	0	0
ACED	53	53	0	0	0
ACSD	471	471	0	0	0
AEDN	584	584	0	0	0
AEUN	10	10	0	0	0
CDD	47	47	0	0	0
CEU	52	50	2	0	2
CEUA	263	263	0	0	0
CEUO	13	13	0	0	0
CIS	2	2	0	0	0
CUC	139	139	0	0	0
EAU	55	55	0	0	0
UAU	1	1	0	0	0
ACATS	1	1	0	0	0
ACSTA	336	336	0	0	0
DPC	52	52	0	0	0
PDC	363	363	0	0	0
PDUP	1	0	1	0	0
PDD	56	0	56	0	0
PDU	5	1	4	0	0
PRU	6	0	6	0	2
PRUP	2	0	2	0	0
PRD	46	0	46	0	22
TOTAL	2746	2629	117	0	26
ESCORE DE MUTAÇÃO	0,96				0,97

Conforme mostrado na Tabela 5.14, foi obtido um total de 2746 mutantes e um escore de mutação de 0,96, dado que somente 26 mutantes foram classificados como equivalentes à ontologia original após a aplicação do conjunto de 1699 dados de teste gerados pelos operadores ACATO, ACOTA, AEDN, AEUN, CEUA, CEUO, ACATS e ACSTA.

Ao analisar os mutantes que permaneceram vivos e não foram classificados como equivalentes dos operadores de mutação PDUP, PDD, PDU, PRU, PRUP e PRD, foi revelado o defeito de "declarar elementos na ontologia e não implementá-los". Como ocorreu nas outras ontologias onde este defeito foi revelado, algumas propriedades foram definidas e não foram utilizadas na representação de nenhum conceito. Ao subtrair o total de mutantes equivalentes do total de mutantes gerados é obtido o escore de mutação de 0,97 ao invés de 0,96.

5.2.3.5 Ontologia Ciclo de Vida de Parasitas - OPL

A Ontologia Ciclo de Vida de Parasitas - OPL (do inglês, *Ontology for Parasite LifeCycle*)[15], é uma ontologia de referência para os estágios do ciclo de vida de parasitas. Esta ontologia possui expressividade lógica $\mathcal{ALCHUE}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, universal, intersecção, união, equivalências e disjunção. Contém 1306 axiomas, sendo 886 axiomas lógicos, 361 classes, 12 propriedades de objetos, 15 indivíduos e 32 propriedades de anotações. A Tabela 5.15 apresenta os operadores de mutação aplicados nesta ontologia, o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes à ontologia original para cada operador de mutação.

Tabela 5.15: Resultados do teste de mutação da Ontologia OPL (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACATO	115	22	0	93	0
ACOTA	40	37	0	3	0
ACED	131	129	0	2	0
ACSD	670	670	0	0	0
AEDN	664	664	0	0	0
CDD	105	105	0	0	0
CDU	45	45	0	0	0
CEU	131	131	0	0	0
CEUA	230	146	0	84	0
CEUO	95	92	0	3	0
CIS	13	13	0	0	0
CUC	302	302	0	0	0
EAU	138	138	0	0	0
UAU	201	201	0	0	0
ACATS	201	201	0	0	0
ACSTA	138	138	0	0	0
DPC	131	131	0	0	0
PDC	360	360	0	0	0
IDU	5	0	5	0	5
PDD	17	9	8	0	0
PDU	6	0	6	0	2
PRU	6	0	6	0	3
PRUP	6	0	6	0	3
PRD	17	6	11	0	3
TOTAL	3767	3540	42	185	16
ESCORE DE MUTAÇÃO	0,98				0,99

Conforme mostrado na Tabela 5.15 foi obtido um total de 3767 mutantes e um escore de mutação de 0,98, dado que somente 16 mutantes foram classificados como equivalentes à ontologia original após a aplicação do conjunto de 1671 dados de teste gerados pelos operadores ACATO, ACOTA, AEDN, CEUA, CEUO, ACATS e ACSTA.

[15]http://www.obofoundry.org/ontology/opl.html

Cmo ocorreu na ontologia OOSTT (Subseção 5.2.3.4), os mutantes que permaneceram vivos e não foram classificados como equivalentes dos operadores de mutação PDD, PDU, PRU, PRUP e PRD, permitiram revelar o defeito de "declarar elementos na ontologia e não implementá-los". Da mesma forma como ocorreu nas outras ontologias onde este defeito foi revelado, algumas propriedades foram definidas e não foram utilizadas na representação de nenhum conceito da ontologia. O operador de mutação ACATO apresentou aproximadamente 80% dos mutantes gerados como mutantes inconsisntes, processo similar ocorreu com o operador de mutação CEUA, onde aproximadamente 35% dos mutantes gerados também foram classificados como inconsistentes. A ocorrência desses mutantes, incluindo também os mutantes dos operadores ACOTA e ACED, se tornaram inconsistentes devido a correta representação de disjunção entre as classes representadas pelo axioma que sofreu a mutação. Após a mutação o axioma se tornava incompatível com a definição de disjunção para ser processado pelo classificador.

A aplicação do teste de mutação nesta ontologia possibilitou revelar a existência de um tipo de defeito[16], conforme é apresentado a seguir:

- declarar elementos na ontologia e não implementá-los.

Considerando que somente 16 mutantes foram classificados como equivalentes, ao subtraí-los do total de mutantes gerados é obtido o escore de mutação de 0,99 ao invés de 0,98.

5.2.3.6 Ontologia Prescrição de Medicamentos - PDRO

A Ontologia Prescrição de Medicamentos - PDRO (do inglês, *The Prescription of Drugs Ontology*)[17], é uma ontologia para descrever entidades relacionadas à prescrição de medicamentos. Esta ontologia possui expressividade lógica $\mathcal{ALNUHE}$, consistindo basicamente de axiomas lógico-descritivos de restrição existencial, universal, intersecção, união, cardinalidades e equivalências. Contém 547 axiomas, dos quais 301 são axiomas lógicos, 212 são classes, 20 são propriedades de objetos, 2 são propriedades de tipos de dados e 12 são propriedades de anotações. Para essa ontologia foi possível a aplicação dos operadores de mutação ACATO, ACOTA, ACED, ACSD, ACATS, ACSTA, AEDN, CDD, CEU, CEUA, CEUO, CUC, ACEMa, ACEMi, ECAU, ACMiEx, ACMiMa, MiCAU, DPC e PDC, conforme os axiomas existentes na ontologia. A Tabela 5.16 apresenta o total de mutantes gerados, mortos, vivos, inconsistentes e equivalentes à ontologia original para cada operador de mutação aplicado.

Tabela 5.16: Resultados do teste de mutação da Ontologia PDRO (autor, 2019)

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
ACATO	33	33	0	0	0
ACOTA	9	9	0	0	0
ACED	12	12	0	0	0
ACSD	272	272	0	0	0
AEDN	186	186	0	0	0
CDD	45	45	0	0	0
CEU	12	12	0	0	0
CEUA	47	45	2	0	2

Continua na próxima página

[16]De acordo com o novo tipo de defeito apresentado na Seção 5.3.1

[17]http://www.obofoundry.org/ontology/pdro.html

Tabela 5.16 – *Continua da página anterior*

Operadores	**Mutantes**				
	Gerados	**Mortos**	**Vivos**	**Inconsistentes**	**Equivalentes**
CEUO	30	30	0	0	0
CUC	162	162	0	0	0
EAU	63	63	0	0	0
UAU	4	4	0	0	0
ACATS	4	4	0	0	0
ACSTA	98	97	1	0	1
ACEMa	2	2	0	0	0
ACEMi	2	2	0	0	0
ECAU	2	2	0	0	0
ACMiEx	1	1	0	0	0
ACMiMa	1	1	0	0	0
MiCAU	1	1	0	0	0
DPC	12	12	0	0	0
PDC	202	202	0	0	0
TOTAL	1200	1197	3	0	3
ESCORE DE MUTAÇÃO	0,99				1

De acordo com a Tabela 5.16 foi obtido um total de 1200 mutantes e um escore de mutação de 0,99, dado que somente 3 mutantes foram classificados como equivalentes após a aplicação do conjunto de 656 dados de teste gerados pelos operadores ACATO, ACOTA, AEDN, CEUA, CEUO, ACATS, ACSTA, ACEMa, ACEMi, ECAU, ACMiEx, ACMiMa e MiCAU.

Nenhum defeito foi revelado nesta ontologia, os únicos 3 mutantes que permaneceram vivos foram classificados como equivalentes, devido não existirem dados de teste no conjunto de dados de teste gerados que produzissem um resultado diferente da ontologia original. Deste modo, ao subtrair o total de mutantes equivalentes do total de mutantes gerados é obtido o escore de mutação de 1 ao invés de 0,99.

5.2.4 Análise dos resultados do Experimento II

Conforme no experimento I (Seção 5.1), para cada ontologia selecionada na atividade (i) (Subseção 5.2.2), também foi proposto aplicar todos os operadores de mutação da Seção 3.4 durante os testes da atividade (ii). Porém, os operadores de mutação utilizados em cada uma das ontologias deste experimento também foram selecionados de acordo com os tipos de axiomas definidos em cada uma das ontologias, de modo que os operadores utilizados neste experimento também não são necessariamente os mesmos para todas as ontologias, variando de acordo com os axiomas definidos em cada uma delas. Todas as ontologias foram selecionadas do repositório *OBO Foundry*, de modo que apresentassem a maior quantidade possível de axiomas compatíveis com os operadores de mutação, sendo utilizados 14 operadores estruturais e 21 operadores lógicos, conforme é apresentado na Tabela 5.17.

Tabela 5.17: Operadores de mutação utilizados em cada ontologia (autor, 2019)

Operadores Estruturais	**Ontologias**					
	CDAO	**IDO**	**ONTONEO**	**OOSTT**	**OPL**	**PDRO**
ACED	X	X	X	X	X	X
ACSD	X	X	X	X	X	X
CDD	X	X	X	X	X	X
CDU	X	X	X		X	
CEU	X	X	X	X	X	X
CUC	X	X	X	X	X	X
PDD	X	X	X	X	X	
PDU	X	X	X	X	X	
PDUP	X	X		X		
PRU	X	X	X	X	X	
DPC	X	X	X	X	X	X
PDC	X	X	X	X	X	X
PRD	X	X	X	X	X	
PRUP	X	X		X	X	
Operadores Lógicos	**Ontologias**					
	CDAO	**IDO**	**ONTONEO**	**OOSTT**	**OPL**	**PDRO**
ACATO	X	X	X	X	X	X
ACOTA	X	X	X	X	X	X
AEDN	X	X	X	X	X	X
AEUN	X	X	X	X		
CEUA	X	X	X	X	X	X
CEUO	X	X	X	X	X	X
CIS	X	X	X	X	X	
ACATS	X	X	X	X	X	X
ACSTA	X	X	X	X	X	X
ECAU	X		X			X
IDU		X			X	
ACEMa	X		X			X
ACEMi	X		X			X
ACEMaEx	X					
ACEMiEx	X		X			X
ACEMaMi	X					
ACEMiMa	X		X			X
MaCAU	X					
MiCAU	X		X			X
EAU	X	X	X	X	X	X
UAU	X	X	X	X	X	X

Os operadores de mutação ISU, ACFT e ACTF não foram utilizados neste experimento por não existirem indivíduos definidos como equivalentes e propriedades de tipos de dados *booleanos* nas ontologias que justificassem a aplicação desses operadores de mutação. O operador de mutação ISU está associado a dois tipos de defeitos, alto grau de especialização e utilizar elementos incorretamente. O defeito utilizar elementos incorretamente foi revelado durante a aplicação do teste de mutação pelo operador ACSD. Porém, o operador de mutação ACSD

não seria capaz de revelar esse defeito caso ele ocorresse na definição de um indivíduo, o mesmo ocorreria para revelar o defeito alto grau de especialização, que não foi revelado por nenhum operador de mutação. Já os operadores ACFT e ACTF estão associados ao defeito definição formal idêntica de classes e instâncias. Esse defeito foi revelado na aplicação do teste de mutação por quatro operadores de mutação estrutural e por dois operadores lógicos de mutação. Porém, como nenhum dos operadores que revelou o defeito definição formal idêntica de classes e instâncias aplica algum tipo de mutação em propriedades de dados, caso houvesse na ontologia em teste um indivíduo representado por uma propriedade de dados do tipo *booleana*, esse defeito não seria revelado pelo teste de mutação. Deste modo, justifica-se a relevância no uso desses três operadores na aplicação do teste de mutação em ontologias que possuam indivíduos definidos na representação do domínio. Somente os operadores de mutação estrutural ACED, ACSD, CDD, CEU, CUC, DPC e PDC, e os operadores lógicos de mutação ACATO, ACOTA, AEDN, CEUA, CEUO, ACATS, ACSTA, EAU e UAU, foram utilizados em todas as ontologias deste experimento. Porém, de todos esses operadores, somente o operador de mutação CEUA possibilitou revelar a existência dos defeitos de uso equivocado de qualificadores de restrição, falta de domínio ou intervalo nas propriedades e assumir características implícitas no nome dos componentes. Já os operadores ACSD, CUC, PDC, ACOTA, AEDN, CEUO e ACSTA, mesmo não revelando defeitos neste experimento, revelaram defeitos no Experimento I e produziram somente 9 mutantes inconsistentes, permitindo deste modo a avaliação dos dados de teste em revelar os defeitos simulados por estes operadores de mutação. Dos 35 operadores de mutação aplicados nas 6 ontologias testadas, 11 revelaram defeitos, conforme pode ser observado na Tabela 5.18.

Tabela 5.18: Operadores de mutação que revelaram defeitos por ontologia (autor, 2019)

Operadores Estruturais	**Ontologias**					
	CDAO	**IDO**	**ONTONEO**	**OOSTT**	**OPL**	**PDRO**
PDD		X	X	X	X	
PDU	X	X	X	X	X	
PDUP	X	X		X		
PRU	X	X	X	X	X	
PRD	X	X	X	X	X	
PRUP	X	X		X	X	
Operadores Lógicos	**Ontologias**					
	CDAO	**IDO**	**ONTONEO**	**OOSTT**	**OPL**	**PDRO**
CEUA		X				
IDU		X				
ACEMa			X			
ACEMi			X			
ECAU			X			

Os operadores ACEMa, ACEMi e ECAU possibilitaram revelar defeitos somente na ontologia ONTONEO. Porém, esses operadores foram aplicados em três ontologias, conforme a Tabela 5.17, de modo que ao todo os operadores ACEMa e ACEMi produziram somente 20 mutantes e o operador ECAU 16, devido a baixa representação de conceitos com axiomas de cardinalidades. Os operadores ACATO, ACOTA, CEUA, CEUO e ACED, produziram mutantes inconsistentes na ontologia OPL que não permitem avaliar os dados de teste. Apesar desses mutantes não revelarem um tipo de defeito nessa ontologia, eles permitiram identificar a correta representação da disjunção entre as classes que sofreram mutação por esses operadores.

Conforme mencionado na Subseção 5.2.3.5, devido a representação de disjunção entre as classes, ao aplicar a mutação no axioma de uma dessas classes o axioma mutado invalidava de forma lógica a ontologia, gerando um mutante inconsistente.

Ao todo foram identificados 7 tipos de defeitos cometidos pelos desenvolvedores nas ontologias testadas, dos quais 6 fazem parte da lista de defeitos apresentada na Subseção 2.3.2, sendo os defeitos intervalo de dados ou domínio limitados, assumir características implícitas no nome dos componentes, falta de domínio ou intervalo nas propriedades, redundância gramatical e uso equívocado de qualificadores de restrição. O defeito declarar elementos na ontologia e não implementá-los, identificado nas 5 ontologias que apresentaram defeitos, não é apresentado na lista de defeitos da Subseção 2.3.2, conforme mencionado na Subseção 5.1.4, sendo classificado como um novo tipo de defeito identificado pelos operadores de mutação PDD, PDU, PDUP, PRU, PRD e PRUP, todos relacionados a mutações realizadas em propriedades de objetos ou de tipos de dados, pelo fato de terem sido criadas propriedades nessas ontologias que não foram utilizadas na representação de nenhum conceito.

Os conjuntos de dados de teste gerados para cada uma das ontologias apresentaram resultados satisfatórios diante do escore de mutação obtido, sendo que após a definição dos mutantes equivalentes o conjunto de dados de teste da ontologia CDAO apresentou escore de mutação acima de 80% e, para as demais ontologias todos acima de 90%. Esses resultados demonstram a adequação dos dados de teste em identificar os defeitos simulados pelos operadores de mutação e auxiliar na avaliação das ontologias diante da ausência de defeitos.

5.3 ANÁLISE DOS RESULTADOS

Nos dois experimentos realizados, Experimento I (Seção 5.1) e Experimento II (Seção 5.2), foram aplicados 35 operadores de mutação de um total de 38 operadores propostos, não sendo utilizados somente os operadores ISU, ACFT e ACTF por não existirem elementos nas ontologias compátiveis com as mutações realizadas por estes operadores. Desses 35 operadores que foram aplicados nos dois experimentos, 19 operadores de mutação possibilitaram revelar defeitos cometidos durante o desenvolvimento das ontologias testadas, conforme pode ser observado na Tabela 5.19, onde são apresentados esses defeitos e quais operadores possibilitaram revelá-los.

Tabela 5.19: Defeitos revelados por operadores de mutação (autor, 2019)

Defeitos Revelados	**Operadores Estruturais**	**Operadores Lógicos**
Intervalo de dados ou domínio limitados	ACSD	
Utilizar elementos incorretamente	ACSD	
Definição formal idêntica de classes e instâncias	ACSD, CDU, CUC, PDC	AEDN, ACSTA
Assumir características implícitas no nome dos componentes	ACSD, PDC	AEDN, ACSTA, CEUA, IDU
Baixo grau de especialização	CDU, CUC	
Utilização incorreta dos operandos AND e OR	CDU	ACOTA, AEDN, CEUO
Declarar elementos na ontologia e não implementá-los	PDD, PDU, PDUP, PRU, PRD, PRUP	
Redundância gramatical	PDC	

Continua na próxima página

Tabela 5.19 – *Continua da página anterior*

Defeitos Revelados	Operadores Estruturais	Operadores Lógicos
Utilizar diferentes critérios de nomenclatura	PDC	
Falta de disjunção		ACOTA
Intervalo de dados ou domínio limitados		AEDN, PDU, PRU
Utilizar elementos incorretamente		AEDN
Uso equivocado de qualificadores de restrição		CEUA
Falta de domínio ou intervalo nas propriedades		CEUA
Redundância gramatical		ACEMa, ACEMi, ECAU
MinIsZero		ACEMi

Conforme mostrado na Tabela 5.19, somente o operador de mutação CEUO não revelou um tipo de defeito diferente ao qual está associado. Já os operadores CDU, ACSTA, CEUA, IDU, PDD, PDUP, PRD, PRUP, ACEMa, ACEMi e ECAU, não revelaram nenhum defeito aos quais estão associados, revelando somente defeitos associados a outros operadores de mutação. Por outro lado, os operadores ACSD, CUC, PDC, AEDN, ACOTA, CEUO, PDU e PRU, além de revelarem defeitos aos quais estão associados, também possibilitaram revelar defeitos associados a outros operadores de mutação. Com base nestes resultados é possível responder as questões de pesquisa 1 e 2, de que os defeitos apresentados na Seção 2.3.2 podem ser revelados com a aplicação do conjunto de operadores de mutação propostos nesta tese e apresentados na Seção 3.4, e de que os operadores de mutação também podem revelar defeitos associados a outros operadores. Como os defeitos da Seção 2.3.2 estão associados a no mínimo um operador de mutação, mesmo que somente 19 dos 35 operadores aplicados nos dois experimentos tenham revelado defeitos nas ontologias testadas, os dados de teste aplicados identificaram a existência do defeito simulado por cada um dos operadores de mutação, indicando a inexistência desse defeito na ontologia original.

Através da análise dos mutantes vivos dos operadores CDU, CUC, PDD, PDU, PDUP, PRU, PRD e PRUP, foi identificada a existência de dois tipos de defeitos que não constam na lista da Seção 2.3.2. Esses defeitos são apresentados na Subseção 5.3.1 e demonstram que o conjunto de operadores proposto é capaz de revelar outros tipos de defeitos que não tenham sido identificados na literatura, respondendo deste modo a questão de pesquisa 3.

Com relação a questão de pesquisa 4, através da análise do escore de mutação de cada uma das ontologias, é possível afirmar que os dados de teste gerados pelo método proposto nesta tese são capazes de identificar os defeitos simulados por cada um dos operadores de mutação. Somente a ontologia OTE do Experimento I teve um baixo escore de mutação, porém isso ocorreu devido a existência de 5 tipos de defeitos nessa ontologia que foram identificados pela análise dos mutantes que permaneceram vivos. Já a ontologia OCP do Experimento I apresentou um escore de mutação de 77% e a ontologia CDAO do Experimento II apresentou um escore de mutação de 84%. Todas as demais ontologias tiveram escores de mutação acima de 90%, demonstrando a adequação dos dados de teste em identificar os defeitos simulados pelos operadores.

Nos dois experimentos realizados foram identificados defeitos nas ontologias, sendo 10 tipos de defeitos identificados em 4 ontologias no Experimento I e, 7 tipos de defeitos identificados em 5 ontologias no Experimento II, o que responde a questão de pesquisa 5. Observa-se que

para a correção desses defeitos é necessária a análise de um especialista do domínio abordado na ontologia, afim de representar de modo correto a definição de cada conceito. Salienta-se também a análise do especialista nos mutantes equivalentes, visto que somente foram utilizados os dados de teste gerados automaticamente, não sendo descartada a hipótese de geração de dados de teste manuais que identifiquem a alteração produzida pelo operador de mutação, ou a identificação de um defeito pelo especialista.

5.3.1 Novos tipos de defeitos

Após a aplicação do teste de mutação para ontologias OWL nos dois experimentos realizados, foram identificados dois tipos de defeitos que não constam na lista dos possíveis defeitos ocorridos durante o desenvolvimento de ontologias apresentada na Subseção 2.3.2. Os dois novos defeitos revelados são apresentados a seguir:

- **Defeito baixo grau de especialização:** este defeito é o contrário do defeito "alto grau de especialização", ele ocorre devido a ontologia estar especializada de tal modo que os "nós" finais da hierarquia devam ser definidos como indivíduos e não como classes.

  Exemplo: Criar uma classe *Carro* e ao invés de criar os tipos de carros como instâncias dessa classe, por serem a menor unidade conceitual e não puderem ser decompostos em mais subtipos, são criadas subclasses para representar cada tipo de carro, de modo que essas subclasses nunca serão instanciadas, ou caso contrário, serão instanciadas por indivíduos que não deveriam pertencer à elas.

- **Defeito declarar elementos na ontologia e não implementá-los:** este defeito ocorre quando são definidos elementos na ontologia, como propriedades de objetos, propriedades de tipos de dados, indivíduos ou classes, mas não são utilizados para representar nenhum conceito do domínio abordado na ontologia.

  Exemplo: Criar propriedades de objetos ou propriedades de tipos de dados e não utilizá-las para representar a associação entre os indivíduos ou para a definição dos atributos de cada objeto representado na ontologia. O mesmo pode ocorrer ao definir um conjunto de indivíduos mas não instanciá-los para nenhuma classe.

5.4 CONSIDERAÇÕES DO CAPÍTULO

Neste capítulo foram apresentados os dois experimentos realizados para avaliar os operadores de mutação, os dados de teste e a aplicação do teste de mutação para ontologias OWL. Nos dois experimentos foram identificados defeitos, sendo 10 tipos de defeitos identificados no Experimento I e 7 tipos de defeitos identificados no Experimento II. Essa comparação possibilitou avaliar que o método de teste proposto é capaz de identificar defeitos ocasionados no desenvolvimento de ontologias por desenvolvedores inexperientes e por especialistas no domínio e especialistas em ontologias. De um modo geral, desenvolvedores inexperientes construirão ontologias mais simples com possibilidades de apresentarem mais defeitos, o que por um lado pode facilitar a identificação desses defeitos e a aplicação do teste em geral. Já especialistas do domínio e especialistas em ontologias construirão ontologias mais complexas, com menores possibilidades da ocorrência de defeitos, o que por outro lado pode dificultar a identificação desses defeitos e a aplicação do teste em geral.

Com base na análise dos resultados de cada um dos experimentos foi possível avaliar a adequação dos operadores de mutação e dos dados de teste, de acordo com os escores de mutação obtidos após a análise dos mutantes vivos em cada ontologia testada. Mesmo que alguns operadores utilizados não tenham revelado defeitos, isso demonstra a inexistência desses defeitos na ontologia original, pelo fato de que os dados de teste identificaram as modificações produzidas por esses operadores, validando deste modo o método de geração de dados de teste proposto.

6 CONSIDERAÇÕES FINAIS

As ontologias são compostas por restrições e axiomas que possibilitam descrever formalmente os conceitos, relacionamentos e propriedades do domínio abordado. Esses elementos que constituem uma ontologia são fundamentais para a criação de uma estrutura que representa o conhecimento de um domínio. No entanto, para criar uma ontologia, o enfoque do problema é a definição para representar e expressar um conceito. O estabelecimento de um conceito pode ter diferentes significados e diferentes modos de representação através da definição de axiomas de acordo com o universo de discurso. Conforme Poveda-Villalón et al. (2014), essa variedade de representações de um domínio pode auxiliar na ocorrência de defeitos durante o desenvolvimento de ontologias, que possam ocasionar falhas ou erros inesperados. Para Hendler (2001), ontologias contextualizadas para diferentes domínios são desenvolvidas localmente por diferentes profissionais e não necessariamente por especialistas em ontologias, que de acordo com Rector et al. (2004) e Poveda-Villalón et al. (2010), os defeitos em sua maioria estão relacionados ao desconhecimento pelo desenvolvedor da ontologia com relação a utilização incorreta de operadores lógicos, enganos em decisões de modelagem, inferências lógicas e definição dos requisitos da ontologia. Por outro lado, defeitos também podem ser inseridos pelo desenvolvedor, que mesmo tendo conhecimento sobre métodos e técnicas de modelagem e construção de ontologias, pode não conhecer padrões de defeitos que possam ser ocasionados.

Diante disso, foi realizado um levantamento dos possíveis defeitos conhecidos que possam ser cometidos durante o desenvolvimento de ontologias, sendo catalogados 57 tipos de defeitos. Esses defeitos foram utilizados como base para o desenvolvimento dos operadores de mutação, de modo que todos os operadores propostos estão associados a um grupo de possíveis defeitos que possam ser revelados pelas mutações realizadas por estes operadores. Ao todo foram propostos 38 operadores de mutação, que foram divididos em 2 grupos. O primeiro grupo corresponde aos operadores de mutação estrutural, composto por 14 operadores que realizam mutações na estrutura de classes, propriedades e indivíduos. O segundo grupo corresponde aos operadores lógicos de mutação, composto por 24 operadores que realizam mutações no conjunto de axiomas que descrevem formalmente os conceitos, relacionamentos e propriedades do domínio.

Para a aplicação do teste de mutação em ontologias OWL foi proposto um método para a geração dos dados de teste, composto por consultas formalizadas em lógica descritiva e representadas pelos axiomas da ontologia em teste, os quais ao sofrerem uma mutação, dão origem a novos dados de teste.

Dois experimentos foram realizados com o objetivo de avaliar a aplicação do teste de mutação para ontologias OWL com relação aos operadores de mutação propostos, a adequação dos dados de teste em revelar os defeitos simulados pelos operadores de mutação e o número de defeitos revelados nas ontologias testadas. Dos 38 operadores de mutação propostos, 35 foram aplicados nos dois experimentos que consistiram de 6 ontologias em cada um. Ao todo foram revelados 13 diferentes tipos de defeitos através das mutações realizadas por 19 operadores de mutação. Para os tipos de defeitos que não foram revelados nas ontologias testadas, pode-se considerar que estes defeitos não foram revelados devido não terem sido cometidos durante o desenvolvimento das ontologias, portanto, estas ontologias estão livres desses defeitos. Já para os operadores de mutação que não revelaram nenhum tipo de defeito, considera-se a adequação dos dados de teste gerados pelo método proposto em identificar os defeitos simulados pelos

operadores de mutação, e indicar a inexistência daquele defeito na ontologia com base no tipo de mutação realizada.

De acordo com os escores de mutação obtidos em cada ontologia dos experimentos realizados, foi possível avaliar a qualidade dos dados de teste gerados e também do método de geração proposto. Das 6 ontologias testadas no Experimento I, uma ontologia apresentou escore de mutação de 77%, uma de 84% e outras 3 acima de 90%. Somente uma ontologia do Experimento I apresentou um baixo escore de mutação, de 28%, devido terem sido identificados 5 tipos de defeitos nessa ontologia após a análise dos mutantes vivos. Já no Experimento II, uma ontologia apresentou escore de mutação de 84% e todas as demais acima de 95%, sendo uma delas com escore de mutação de 100%. Esses resultados demonstram a qualidade dos dados de teste utilizados nestes experimentos em identificar os defeitos simulados pelos operadores de mutação, e consequentemente validam o método de geração dos dados de teste.

6.1 CONTRIBUIÇÕES

Além das contribuições desta tese mencionadas na Seção 1.4, destacam-se também:

- Tanto os operadores de mutação estrutural quanto os operadores lógicos de mutação revelaram defeitos nas ontologias testadas. Isso mostra que ambos os tipos de mutação são necessários no teste de mutação para ontologias;
- dois novos tipos de defeitos não identificados na lista de defeitos apresentada na Subseção 2.3.2 foram revelados, o que mostra que o teste proposto apresenta uma possibilidade em revelar defeitos não previstos durante a elaboração dos operadores de mutação e dos dados de teste; e
- através da análise dos mutantes inconsistentes, que mesmo não permitindo avaliar os dados de teste e portanto não são contabilizados no cálculo do escore de mutação, é possível identificar a existência de defeitos, conforme ocorreu na ontologia OCP do Experimento I.

6.2 LIMITAÇÕES

Consideram-se as seguintes limitações desta tese em relação a aplicação do teste de mutação para ontologias:

- A utilização de duas ferramentas para a aplicação do teste, sendo a ferramenta MutaOnto para geração dos mutantes e dos dados de teste proposta nesta tese e, a ferramenta *Protégé* para a aplicação dos dados de teste na ontologia original e nas ontologias mutantes, por falta de recursos para a execução desta atividade na ferramenta MutaOnto;
- a seleção manual dos dados de teste após a geração desses dados pela ferramenta MutaOnto, para a aplicação na ferramenta *Protégé*, pela ausência de um método automatizado de seleção e aplicação dos dados de teste;
- a necessidade da execução individual de cada ontologia mutante pelo motivo de a ferramenta *Protégé* não permitir a execução simultânea de várias ontologias; e
- a falta de recursos na ferramenta MutaOnto, como a implementação de *reasoners* que possibilitem a análise da ontologia nesta ferramenta, e também a execução de várias ontologias simultaneamente, o que permitirá reduzir o tempo de aplicação do teste.

6.3 TRABALHOS FUTUROS

Como trabalhos futuros decorrentes desta tese, destacam-se:

- A replicação dos experimentos realizados, considerando ontologias de outros domínios, com um número maior de indivíduos instanciados e que seja possível a aplicação dos 3 operadores de mutação que não foram aplicados nos experimentos realizados;
- replicar os experimentos utilizando métodos de mutação seletiva, com o objetivo de otimizar o número de mutantes gerados e reduzir o custo de aplicação do teste;
- implementar um *reasoner* na ferramenta MutaOnto, para que seja possível a análise lógica das inferências da ontologia pela própria ferramenta e, um mecanismo de execução de consultas em várias ontologias simultaneamente. Isso possibilitará que não seja mais necessária a utilização da ferramenta *Protégé*, melhorando o tempo de execução do teste de mutação para ontologias;
- implementar na ferramenta MutaOnto duas estruturas de visualização do código fonte da ontologia, uma para mostrar o código fonte da ontologia original e outra para mostrar a mutação realizada em cada ontologia mutante;
- adaptar o método de geração de dados de teste para a linguagem de consultas para ontologias SPARQL, com o objetivo de comparar os resultados com o método proposto nesta tese, com relação ao número de mutantes mortos, vivos e equivalentes.

REFERÊNCIAS

610.12, I. (1990). IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*.

Abburu, S. (2012). A survey on ontology reasoners and comparison. *International Journal of Computer Applications*, 57(17):33–39.

Agrawal, H., DeMillo, R. A., Hathaway, R., Hsu, W., Krauser, E. W., Martin, R. J., Mathur, A. P. e Spafford, E. H. (1989). Design of mutant operators for the c programming language. Relatório Técnico SERC-TR41-P, Purdue University, West Lafayette, IN, EUA.

Allemang, D. e Hendler, J. (2011). *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Elsevier Inc., Waltham, MA, USA.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. e Patel-Schneider, P. F., editores (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Baader, F., Horrocks, I. e Sattler, U. (2008). *Handbook of Knowledge Representation*. Elsevier Science, San Diego, USA.

Baader, F. e Nutt, W. (2003). *The Description Logic Handbook*. Cambridge University Press, New York, NY, USA.

Bartolini, C. (2016). Mutation owls: semantic mutation testing for ontologies. Em *Proceedings of the International Workshop on domain specific Model-based approaches to verification and validation*, páginas 43–53, Rome - Italy.

Baumeister, J. e Seipel, D. (2005). Smelly owls – design anomalies in ontologies. Em *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference*, páginas 215–220, Florida - USA.

Baumeister, J. e Seipel, D. (2010). Anomalies in ontologies with rules. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(1).

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F. e Stein, L. A. (2004). Owl web ontology language reference. `https://www.w3.org/TR/owl-ref/`. Acessado em 15/02/2017.

Berners-Lee, T., Hendler, J. e Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):28–37.

Bezerra, C., Santana, F. e Freitas, F. (2014). Cqchecker: A tool to check ontologies in owl-dl using competency questions written in controlled natural language. *Learning & Nonlinear Models*, 12(2):115–129.

Blomqvist, E., Sepour, A. S. e Presutti, V. (2012). Ontology testing-methodology and tool. Em *18th International Conference in Knowledge Engineering and Knowledge Management*, páginas 216–226, Galway City - Ireland.

Borst, W. N. (1997). *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. Tese de doutorado, Universiteit Twente, Enschede.

Brank, J., Grobelnik, M. e Mladenić, D. (2005). A survey of ontology evaluation techniques. Em *Proceedings of 8th International Multi-Conference of the Information Society*, páginas 166–169.

Budd, T. A. (1980). *Mutation Analysis of Program Test Data*. Tese de doutorado, Yale University, Yale, NH - USA.

Budd, T. A. (1981). Mutation analysis: Ideas, examples, problems and prospects. Em *Computer Program Testing*, páginas 129–148, Amsterdam - North Holland.

Cabeça, A. G., Jino, M. e Leitão-Junior, P. S. (2009). Mutation analysis for sql database applications. Em *The Fourth International Conference on Software Engineering Advances - ICSEA*, páginas 146–151, Porto, Portugal.

Copeland, M. (2016). Version analysis for fault detection in owl ontologies. Dissertação de Mestrado, Faculty in Engineering and Physical Sciences - University of Manchester, Manchester - UK.

Copeland, M., Gonçalves, R. S., Parsia, B., Sattler, U. e Stevens, R. (2013). Finding fault: detecting issuesin a versioned ontology. Em *European Semantic Web Conference - ESWC*, páginas 113–124, Montpellier, France.

Corcho, O., Gómez-Pérez, A., González-Cabero, R. e Suárez-Figueroa, M. C. (2004). Odeval: a tool for evaluating rdf(s), daml+oil, and owl concept taxonomies. Em *1st International Conference on Artificial Intelligence Applications and Innovations - AIAI*, Toulouse - Fance.

Corcho, O., Roussey, C. e Blazquez, L. M. V. (2009). Catalogue of anti-patterns for formal ontology debugging. Em *Atelier Construction d'ontologies: vers un guide des bonnes pratiques - AFIA*, páginas 2–12, Hammamet - Tunisie.

Davies, J., Fensel, D. e van Harmelen, F. (2003). *Semantic Web: Ontology-driven Knowledge Management*. John Wiley & Sons Ltd.

Delamaro, M. E., Maldonado, J. C. e Jino, M. (2007). *Introdução ao teste de software*. Campus. Elsevier, Rio de Janeiro.

DeMillo, R. A., Lipton, R. J. e Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

DeMillo, R. A. e Offut, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910.

Dentler, K., Cornet, R., ten Teije, A. e de Keizer, N. (2011). Comparison of reasoners for large ontologies in the owl 2 el profile. *Semantic Web*, 2(2):71–87.

Derezinska, A. (2003). Object-oriented mutation to assess the quality of tests. Em *Proceedings of the 29th Euromicro Conference*, páginas 417–420, Belek - Turkey.

Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T. e Masiero, P. C. (1999). Mutation testing applied to validate speci
cations based on statecharts. Em *Proceedings of the 10th International Symposium on Software Reliability Engineering*, páginas 210–219, Boca Raton, Florida - USA.

Freitas, F. e Schulz, S. (2009). Ontologias, web semântica e saúde. *Revista Eletrônica de Comunicação, Informação & Inovação em Saúde*, 3(1):4–7.

Gangemi, A., Catenacci, C., Ciaramita, M. e Lehmann, J. (2006). Qood grid: A metaontology-based framework for ontology evaluation and selection. Em *Proceedings of Evaluation of Ontologies for the Web, 4th International EON Workshop, Located at the 15th International World Wide Web Conference*, Edinburgh - UK.

García-Ramos, S., Otero, A. e Fernández-López, M. (2009). Ontologytest: A tool evaluate ontologies through tests defined by the user. Em *10th International Work-Conference on Artificial Neural Networks*, páginas 91–98, Salamanca - Spain.

García-Ramos, S., Otero, A. e Fernández-López, M. (2016). Ontologytest: A tool to evaluate ontologies through tests defined by the user. Em *10th International Work-Conference on Artificial Neural Networkst*, páginas 91–98, Salamanca - Spain.

Gasevic, D., Djuric, D. e Devedzic, V. (2009). *Model Driven Engineering and Ontology Development*. Springer.

Gómez-Pérez, A. (1999). Evaluation of taxonomic knowledge on ontologies and knowledge-based systems. Em *Proceedings of the Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta - Canadá.

Gómez-Pérez, A. (2004). *Ontology Evaluation*, páginas 251–273. Springer Berlin Heidelberg.

Grüninger, M. e Fox, M. S. (1995). Methodology for the design and evaluation of ontologies. Em *Workshop on Basic Ontological Issues in Knowledge Sharing in 14th International Joint Conference on Artificial Intelligence*, Montreal, Quebec - Canada.

Gruber, T. R. (1995). Toward principles for the design of ontologies used fo knowledge sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928.

Guarino, N. (1995). Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human-Computer Studies*, 43(5-6):625–640.

Haendel, M., Lewis, S., Natale, D., Peters, B., Rocca-Serra, P., Schriml, L., Stoeckert, C. e Walls, R. (2019). The obo foundry. `http://www.obofoundry.org/`. Acessado em 17/05/2018.

Hayes, P. J. e Patel-Schneider, P. F. (2014). Rdf 1.1 semantics. w3c recommendation. `http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/`. Acessado em 20/09/2018.

Hendler, J. (2001). Agents and the semantic web. *IEEE Inteligent Systems*, 16(2):30–37.

Hitzler, P., Krötzsch, M. e Rudolph, S. (2010). *Foundations of Semantic Web Technologies*. CRC Press.

Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R. e Wang, H. (2006). The manchester owl syntax. Em *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions*, Athens, GA, USA.

Horrocks, I. (2008). Ontologies and the semantic web. *Communications of the ACM - Surviving the data deluge*, 51(12):58–67.

Howden, W. E. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379.

Jepsen, T. C. (2009). Just what is an ontology, anyway? *IT Professional*, 11(5):22–27.

Jia, Y. e Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

Knublauch, H. (2003). Case study: Using protégé to convert the travel ontology to uml and owl. Em *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools - EON held at the 22nd International Semantic Web Conference - ISWC*, Sanibel Island, Florida - USA.

Korst, J., Geleijnse, G., de Jong, N. e Verschoor, M. (2006). Ontology-based information extraction from the world wide web. *Intelligent Algorithms in Ambient and Biomedical Computing*, 7:149–167.

Lee, S., Bai, X. e Chen, Y. (2008). Automatic mutation testing and simulation on owl-s specified web services. Em *41st Annual Simulation Symposium*, páginas 149–156, Ottawa - Canasa.

Maedche, A. e Staab, S. (2002). Measuring similarity between ontologies. Em *13th International Conference on Knowledge Engineering and Knowledge Management - EKAW. Ontologies and the Semantic Web*, páginas 251–263, Siguenza - Espanha.

Maldonado, J. C., Vincenzi, A. M. R., Barbosa, E. F., do S. Rocio Senger de Souza e Delamaro, M. E. (1998). *Aspectos teóricos e empíricos de teste de cobertura de software*. ICMSC-USP.

McGuinness, D. L. e van Harmelen, F. (2004). Owl web ontology language overview. `http://www.w3.org/TR/owl-features/`. Acessado em 13/02/2017.

Musen, M. A. (2015). The protégé project: A look back and a look forward. *AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence*, 1(4).

Nardon, F. B. (2003). *Compartilhamento de Conhecimento em Saúde utilizando Ontologias e Bancos de Dados Dedutivos*. Tese de doutorado, USP - Universidade de São Paulo, São Paulo - Brasil.

Obrst, L., Ashpole, B., Ceusters, W., Mani, I., Ray, S. e Smith, B. (2007). *The Evaluation of Ontologies: Toward improved semantic interoperability*, páginas 139–158. Springer US.

Porn, A. M. (2014). Teste de mutação para ontologias owl. Dissertação de Mestrado, Pós-Graduação em Informática - Universidade Federal do Paraná, Curitiba - PR.

Porn, A. M., Huve, C. A. G., Peres, L. M. e Direne, A. I. (2016). A systematic literature review of owl ontology evaluation. Em *15th International Conference on WWW/Internet*, páginas 67–74, Mannheim - Alemanha.

Porn, A. M. e Peres, L. M. (2014). Mutation test to owl ontologies. Em *13th International Conference on WWW/Internet*, páginas 123–130, Porto - Portugal.

Porn, A. M. e Peres, L. M. (2017). Semantic mutation test to owl ontologies. Em *19th International Conference on Enterprise Information Systems*, Porto - Portugal.

Porn, A. M., Peres, L. M. e del Fabro, M. D. (2015). A process for the representation of openehr adl archetypes in owl ontologies. Em *MEDINFO 2015: eHealth-enabled Health*, páginas 827–831, São Paulo, Brazil.

Porzel, R. e Malaka, R. (2004). A task-based approach for ontology evaluation. Em *Proceedings of the Workshop on Ontology Learning and Population at the 16th European Conference on Artificial Intelligence*, Valencia - Spain.

Poveda-Villalón, M., Gómez-Pérez, A. e Suárez-Figueroa, M. C. (2014). Oops! (ontology pitfall scanner!): An on-line tool for ontology evaluation. *International Journal on Semantic Web & Information Systems*, 10(2):7–34.

Poveda-Villalón, M., Suárez-Figueroa, M. C. e Gómez-Pérez, A. (2010). A double classification of common pitfalls in ontologies. Em *OntoQual 2010 - Workshop on Ontology Quality at the 17th International Conference on Knowledge Engineering and Knowledge Management - EKAW)*, páginas 1–12, Lisbon - Portugal.

Poveda-Villalón, M., Suárez-Figueroa, M. C. e Gómez-Pérez, A. (2012). Validating ontologies with oops! Em *18th International Conference in Knowledge Engineering and Knowledge Management*, páginas 267–281, Galway City - Ireland.

Prado, S. G. D. (2005). *Um experimento no uso de ontologias para reforço da aprendizagem em educação a distância*. Tese de doutorado, USP - Universidade de São Paulo, São Paulo - Brasil.

Pressman, R. S. (2011). *Engenharia de Software: Uma abordagem profissional*. AMGH, Porto Alegre, RS.

Qazi, N. I. e Qadir, M. A. (2010). Algorithms to evaluate ontologies based on extended error taxonomy. Em *International Conference on Information and Emerging Technologies*, páginas 1–6, Karachi, Pakistan.

Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H. e Wroe, C. (2004). Owl pizzas: Practical experience of teaching owl dl: Common errors& common patterns. Em *14th International Conference on Knowledge Engineering and Knowledge Management*, páginas 63–81, Whittlebury Hall - United Kingdom.

Reiter, R. (1978). On closed world data bases. *Logic and Data Bases*, páginas 55–76.

Rieckhof, F., Dibowski, H. e Kabitzsch, K. (2011). Formal validation techniques for ontology-based device descriptions. Em *16th Conference on Emerging Technologies & Factory Automation -ETFA*, páginas 1–8, Toulouse - France.

Rodes, R. E. e Pospesel, H. (1996). *Premises and Conclusions: Synbolic Logic fo Legal Analysis*. Pearson.

Schmidt-Schauß, M. e Smolka, G. (1991). Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26.

Sequeda, J. F. (2015). *Integrating Relational Databases with the Semantic Web*. Tese de doutorado, The University of Texas at Austin, Austin, Texas.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A. e Katz, Y. (2007). Pellet: A practical owldl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53.

Smith, M. K., Welty, C. e McGuinness, D. L. (2004). Owl web ontology language guide. `https://www.w3.org/TR/owl-guide/`. Acessado em 15/02/2017.

Sousa, L. G. e Leite, J. (2005). Geração automática de dicionários explicativos em sistemas de informações geográficas usando ontologia. Em *XXV Congresso da Sociedade Brasileira de Computação*, páginas 203–212, Porto Alegre - Brasil.

Stojanovic, N., Studer, R. e Stojanovic, L. (2003). An approach for the ranking of query results in the semantic web. *The Semantic Web*, 2870:500–516.

Tuya, J., Suárez-Cabal, M. J. e de la Riva, C. (2007). Mutating database queries. *Information and Software Technology*, 49(4):398–417.

Uschold, M. e Gruninger, M. (1996). Ontologies: Principles, methods and applications. *Knowledge Engineering Reveiw*, 11(2):93–136.

Vrandečić, D. (2010). *Ontology Evaluation*. Tese de doutorado, Fakultät für Wirtschaftswissenschaften des Karlsruher Instituts für Technologie, Karlsruhe.

Vrandečić, D. e Gangemi, A. (2006). Unit tests for ontologies. Em *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, páginas 1012–1020, Montpellier - France.

Wimalasuriya, D. C. e Dou, D. (2009). Using multiple ontologies in information extraction. Em *Proceedings of the 18th ACM conference on Information and knowledge management*, páginas 235–244, Hong Kong - China.

Wong, W. E., Mathur, A. P. e Maldonado, J. C. (1995). Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. Em *Software Quality and Productivity: Theory, practice and training*, páginas 258–265, London - United Kingdom.

APÊNDICE A – OPERADORES DE MUTAÇÃO ESTRUTURAL OWL

A.1 OPERADOR DE MUTAÇÃO CLASSUPCASCADE - CUC

Exemplo A.1: Aplicação do operador de mutação CUC (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Profissional''>
      <rdfs:subClassOf rdf:resource="Pessoas" />
  </owl:Class>

  <owl:Class rdf:about=''Pessoas''>
      <rdfs:subClassOf rdf:resource=''Thing'' />
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Profissional''>
      <rdfs:subClassOf rdf:resource="Thing" />
  </owl:Class>

  <owl:Class rdf:about=''Pessoas''>
      <rdfs:subClassOf rdf:resource=''Thing'' />
  </owl:Class>
```

A.2 OPERADOR DE MUTAÇÃO CLASSDISJOINTDEF - CDD

Exemplo A.2: Aplicação do operador de mutação CDD (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Médico''>
      <rdfs:subClassOf rdf:resource=''Profissional'' />
  </owl:Class>

  <owl:Class rdf:about=''Enfermeiro''>
      <rdfs:subClassOf rdf:resource=''Profissional'' />
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Médico''>
      <rdfs:subClassOf rdf:resource=''Profissional'' />
        <owl:disjointWith rdf:resource=''Enfermeiro'' />
  </owl:Class>
```

A.3 OPERADOR DE MUTAÇÃO CLASSEQUIVALENTUNDEF - CEU

Exemplo A.3: Aplicação do operador de mutação CEU (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Gestante''>
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType=''Collection''>
          <rdf:Description rdf:about=''Pessoas'' />
          <owl:Restriction>
            <owl:onProperty rdf:resource=''éGestante'' />
            <owl:hasValue rdf:datatype=''boolean''>true</owl:hasValue>
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Gestante'' />
```

A.4 OPERADOR DE MUTAÇÃO PROPERTYDOMAINDOWN - PDD

Exemplo A.4: Aplicação do operador de mutação PDD (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Homem''>
      <rdfs:subClassOf rdf:resource=''Pessoas'' />
  </owl:Class>

  <owl:DatatypeProperty rdf:about=''éGestante''>
      <rdfs:domain rdf:resource=''Pessoas'' />
  </owl:DatatypeProperty>

- ONTOLOGIA MUTANTE
  <owl:DatatypeProperty rdf:about=''éGestante''>
      <rdfs:domain rdf:resource=''Homem'' />
  </owl:DatatypeProperty>
```

A.5 OPERADOR DE MUTAÇÃO PROPERTYDOMAINUP - PDUP

Exemplo A.5: Aplicação do operador de mutação PDUP (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Pessoas''>
      <rdfs:subClassOf rdf:resource=''Thing'' />
  </owl:Class>

  <owl:DatatypeProperty rdf:about=''éGestante''>
      <rdfs:domain rdf:resource=''Pessoas'' />
  </owl:DatatypeProperty>

- ONTOLOGIA MUTANTE
  <owl:DatatypeProperty rdf:about=''éGestante''>
      <rdfs:domain rdf:resource=''Thing'' />
  </owl:DatatypeProperty>
```

A.6 OPERADOR DE MUTAÇÃO PROPERTYRANGEDOWN - PRD

Exemplo A.6: Aplicação do operador de mutação PRD (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Feminino''>
      <rdfs:subClassOf rdf:resource=''Sexo'' />
  </owl:Class>

  <owl:ObjectProperty rdf:about=''temSexo''>
      <rdfs:domain rdf:resource=''Pessoas'' />
      <rdfs:range rdf:resource=''Sexo'' />
  </owl:ObjectProperty>

- ONTOLOGIA MUTANTE
  <owl:ObjectProperty rdf:about=''temSexo''>
      <rdfs:domain rdf:resource=''Pessoas'' />
      <rdfs:range rdf:resource=''Feminino'' />
  </owl:ObjectProperty>
```

A.7 OPERADOR DE MUTAÇÃO PROPERTYRANGEUP - PRUP

Exemplo A.7: Aplicação do operador de mutação PRUP (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Sexo''>
      <rdfs:subClassOf rdf:resource=''Thing'' />
  </owl:Class>

  <owl:ObjectProperty rdf:about=''temSexo''>
      <rdfs:domain rdf:resource=''Pessoas'' />
      <rdfs:range rdf:resource=''Sexo'' />
  </owl:ObjectProperty>

- ONTOLOGIA MUTANTE
  <owl:ObjectProperty rdf:about=''temSexo''>
      <rdfs:domain rdf:resource=''Pessoas'' />
      <rdfs:range rdf:resource=''Thing'' />
  </owl:ObjectProperty>
```

A.8 OPERADOR DE MUTAÇÃO PROPERTYDOMAINUNDEF - PDU

Exemplo A.8: Aplicação do operador de mutação PDU (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:ObjectProperty rdf:about=''éGestante''>
      <rdfs:domain rdf:resource=''Pessoas'' />
  </owl:ObjectProperty>

- ONTOLOGIA MUTANTE
  <owl:ObjectProperty rdf:about=''éGestante'' />
```

A.9 OPERADOR DE MUTAÇÃO PROPERTYRANGEUNDEF - PRU

Exemplo A.9: Aplicação do operador de mutação PRU (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:ObjectProperty rdf:about=''temSexo''>
      <rdfs:domain rdf:resource=''Pessoas'' />
      <rdfs:range rdf:resource=''Sexo'' />
  </owl:ObjectProperty>

- ONTOLOGIA MUTANTE
  <owl:ObjectProperty rdf:about=''temSexo'' />
      <rdfs:domain rdf:resource=''Pessoas'' />
  </owl:ObjectProperty>
```

A.10 OPERADOR DE MUTAÇÃO AXIOMCHANGESUBCLASSTODISJOINT - ACSD

Exemplo A.10: Aplicação do operador de mutação ACSD (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Homem''>
      <rdfs:subClassOf>
          <owl:Class>
              <owl:intersectionOf rdf:parseType=''Collection''>
                  <rdf:Description rdf:about=''Pessoas'' />
                  <owl:Restriction>
                      <owl:onProperty rdf:resource=''temSexo'' />
                      <owl:someValuesFrom rdf:resource=''Masculino'' />
                  </owl:Restriction>
              </owl:intersectionOf>
          </owl:Class>
      </rdfs:subClassOf>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Homem''>
      <rdfs:disjointWith>
          <owl:Class>
              <owl:intersectionOf rdf:parseType=''Collection''>
                  <rdf:Description rdf:about=''Pessoas'' />
                  <owl:Restriction>
                      <owl:onProperty rdf:resource=''temSexo'' />
                      <owl:someValuesFrom rdf:resource=''Masculino'' />
                  </owl:Restriction>
              </owl:intersectionOf>
          </owl:Class>
      </rdfs:disjointWith>
  </owl:Class>
```

A.11 OPERADOR DE MUTAÇÃO AXIOMCHANGEEQUIVALENTTODISJOINT - ACED

Exemplo A.11: Aplicação do operador de mutação ACED (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Homem''>
      <rdfs:equivalentTo>
          <owl:Class>
              <owl:intersectionOf rdf:parseType=''Collection''>
                  <rdf:Description rdf:about=''Pessoas'' />
                  <owl:Restriction>
                      <owl:onProperty rdf:resource=''temSexo'' />
                      <owl:someValuesFrom rdf:resource=''Masculino'' />
                  </owl:Restriction>
              </owl:intersectionOf>
          </owl:Class>
      </rdfs:equivalentTo>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Homem''>
      <rdfs:disjointWith>
          <owl:Class>
              <owl:intersectionOf rdf:parseType=''Collection''>
                  <rdf:Description rdf:about=''Pessoas'' />
                  <owl:Restriction>
                      <owl:onProperty rdf:resource=''temSexo'' />
                      <owl:someValuesFrom rdf:resource=''Masculino'' />
                  </owl:Restriction>
              </owl:intersectionOf>
          </owl:Class>
      </rdfs:disjointWith>
  </owl:Class>
```

A.12 OPERADOR DE MUTAÇÃO PRIMITIVETODEFINEDCLASS - PDC

Exemplo A.12: Aplicação do operador de mutação PDC (autor, 2019)

```
- ONTOLOGIA ORIGINAL
    <owl:Class rdf:about=''Homem''>
        <rdfs:SubClassOf>
            <owl:Class>
                <owl:intersectionOf rdf:parseType=''Collection''>
                    <rdf:Description rdf:about=''Pessoas'' />
                    <owl:Restriction>
                        <owl:onProperty rdf:resource=''temSexo'' />
                        <owl:someValuesFrom rdf:resource=''Masculino'' />
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </rdfs:SubClassOf>
    </owl:Class>

- ONTOLOGIA MUTANTE
    <owl:Class rdf:about=''Homem''>
        <rdfs:equivalentTo>
            <owl:Class>
                <owl:intersectionOf rdf:parseType=''Collection''>
                    <rdf:Description rdf:about=''Pessoas'' />
                    <owl:Restriction>
                        <owl:onProperty rdf:resource=''temSexo'' />
                        <owl:someValuesFrom rdf:resource=''Masculino'' />
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </rdfs:equivalentTo>
    </owl:Class>
```

A.13 OPERADOR DE MUTAÇÃO DEFINEDTOPRIMITIVECLASS - DPC

Exemplo A.13: Aplicação do operador de mutação DPC (autor, 2019)

```
- ONTOLOGIA ORIGINAL
    <owl:Class rdf:about=''Homem''>
        <rdfs:equivalentTo>
            <owl:Class>
                <owl:intersectionOf rdf:parseType=''Collection''>
                    <rdf:Description rdf:about=''Pessoas'' />
                    <owl:Restriction>
                        <owl:onProperty rdf:resource=''temSexo'' />
                        <owl:someValuesFrom rdf:resource=''Masculino'' />
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </rdfs:equivalentTo>
    </owl:Class>

- ONTOLOGIA MUTANTE
    <owl:Class rdf:about=''Homem''>
        <rdfs:SubClassOf>
            <owl:Class>
                <owl:intersectionOf rdf:parseType=''Collection''>
                    <rdf:Description rdf:about=''Pessoas'' />
                    <owl:Restriction>
                        <owl:onProperty rdf:resource=''temSexo'' />
                        <owl:someValuesFrom rdf:resource=''Masculino'' />
                    </owl:Restriction>
                </owl:intersectionOf>
            </owl:Class>
        </rdfs:SubClassOf>
    </owl:Class>
```

A.14 OPERADOR DE MUTAÇÃO CLASSDISJOINTUNDEF - CDU

Exemplo A.14: Aplicação do operador de mutação CDU (autor, 2019)

```
- ONTOLOGIA ORIGINAL
    <owl:Class rdf:about=''Feminino''>
        <owl:disjointWith rdf:resource=''Masculino'' />
    </owl:Class>

- ONTOLOGIA MUTANTE
    <owl:Class rdf:about=''Feminino'' />
    <owl:Class rdf:about=''Masculino'' />
```

APÊNDICE B – OPERADORES LÓGICOS DE MUTAÇÃO OWL

B.1 OPERADOR DE MUTAÇÃO CLASSEQUIVALENTUNDEFAND - CEUA

Exemplo B.1: Aplicação do operador de mutação CEUA (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Gestante''>
 <owl:equivalentClass>
     <owl:Class>
         <owl:intersectionOf rdf:parseType=''Collection''>
             <rdf:Description rdf:about=''Pessoas'' />
             <owl:Restriction>
                 <owl:onProperty rdf:resource=''éGestante'' />
                 <owl:hasValue rdf:datatype=''boolean''>true</owl:hasValue>
             </owl:Restriction>
         </owl:intersectionOf>
     </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Gestante''>
    <owl:equivalentClass rdf:resource=''Pessoas'' />
</owl:Class>

<owl:Class rdf:about=''Gestante''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''éGestante'' />
            <owl:hasValue rdf:datatype=''boolean''>true</owl:hasValue>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

B.2 OPERADOR DE MUTAÇÃO CLASSEQUIVALENTUNDEFOR - CEUO

Exemplo B.2: Aplicação do operador de mutação CEUO (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Jovem''>
    <owl:equivalentClass>
        <owl:Class>
            <owl:unionOf rdf:parseType=''Collection''>
                <owl:Restriction>
                    <owl:onProperty rdf:resource=''temCaracteristica'' />
                    <owl:hasValue rdf:resource=''Menina'' />
                </owl:Restriction>
                <owl:Restriction>
                    <owl:onProperty rdf:resource=''temCaracteristica'' />
                    <owl:hasValue rdf:resource=''Menino'' />
                </owl:Restriction>
            </owl:unionOf>
        </owl:Class>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Jovem''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''temCaracteristica'' />
            <owl:hasValue rdf:resource=''Menina'' />
        </owl:Restriction>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
</owl:Class>

<owl:Class rdf:about=''Jovem''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''temCaracteristica'' />
            <owl:hasValue rdf:resource=''Menino'' />
        </owl:Restriction>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
</owl:Class>
```

B.3 OPERADOR DE MUTAÇÃO AXIOMCHANGEANDTOOR - ACATO

Exemplo B.3: Aplicação do operador de mutação ACATO (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Jovem''>
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType=''Collection''>
          <rdf:Description rdf:about=''Jovem'' />
          <owl:Restriction>
            <owl:onProperty rdf:resource=''temCaracteristica'' />
            <owl:hasValue rdf:resource=''Menino'' />
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
</owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Jovem''>
    <owl:equivalentClass>
      <owl:Class>
        <owl:unionOf  rdf:parseType=''Collection''>
          <rdf:Description rdf:about=''Jovem'' />
          <owl:Restriction>
            <owl:onProperty rdf:resource=''temCaracteristica'' />
            <owl:hasValue rdf:resource=''Menino'' />
          </owl:Restriction>
        </owl:unionOf>
      </owl:Class>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
</owl:Class>
```

B.4 OPERADOR DE MUTAÇÃO AXIOMCHANGEORTOAND - ACOTA

Exemplo B.4: Aplicação do operador de mutação ACOTA (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Jovem''>
    <owl:equivalentClass>
      <owl:Class>
        <owl:unionOf rdf:parseType=''Collection''>
          <rdf:Description rdf:about=''Jovem'' />
          <owl:Restriction>
            <owl:onProperty rdf:resource=''temCaracteristica'' />
            <owl:hasValue rdf:resource=''Meninoa'' />
          </owl:Restriction>
        </owl:unionOf>
      </owl:Class>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
</owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Jovem''>
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType=''Collection''>
          <rdf:Description rdf:about=''Jovem'' />
          <owl:Restriction>
            <owl:onProperty rdf:resource=''temCaracteristica'' />
            <owl:hasValue rdf:resource=''Menino'' />
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
</owl:Class>
```

B.5 OPERADOR DE MUTAÇÃO AXIOMCHANGEALLTOSOME - ACATS

Exemplo B.5: Aplicação do operador de mutação ACATS (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
      <owl:equivalentClass>
          <owl:Restriction>
              <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
              <owl:allValuesFrom rdf:resource=''Profissional'' />
          </owl:Restriction>
      </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
      <owl:equivalentClass>
          <owl:Restriction>
              <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
              <owl:someValuesFrom rdf:resource=''Profissional'' />
          </owl:Restriction>
      </owl:equivalentClass>
  </owl:Class>
```

B.6 OPERADOR DE MUTAÇÃO AXIOMCHANGESOMETOALL - ACSTA

Exemplo B.6: Aplicação do operador de mutação ACSTA (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
      <owl:equivalentClass>
          <owl:Restriction>
              <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
              <owl:someValuesFrom rdf:resource=''Profissional'' />
          </owl:Restriction>
      </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
      <owl:equivalentClass>
          <owl:Restriction>
              <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
              <owl:allValuesFrom rdf:resource=''Profissional'' />
          </owl:Restriction>
      </owl:equivalentClass>
  </owl:Class>
```

B.7 OPERADOR DE MUTAÇÃO AXIOMEQUIVALENTDEFNOT - AEDN

Exemplo B.7: Aplicação do operador de mutação AEDN (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
            <owl:allValuesFrom rdf:resource=''Profissional'' />
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:complementOf rdf:resource=''Profissional''>
                </owl:Class>
            <owl:allValuesFrom>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

B.8 OPERADOR DE MUTAÇÃO AXIOMEQUIVALENTUNDEFNOT - AEUN

Exemplo B.8: Aplicação do operador de mutação AEUN (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:complementOf rdf:resource=''Profissional''>
                </owl:Class>
            <owl:allValuesFrom>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
            <owl:allValuesFrom rdf:resource=''Profissional'' />
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

B.9 OPERADOR DE MUTAÇÃO AXIOMCHANGEMINTOMAX - ACMIMA

Exemplo B.9: Aplicação do operador de mutação ACMiMa (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:minQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:minQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:maxQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:maxQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.10 OPERADOR DE MUTAÇÃO AXIOMCHANGEMAXTOMIN - ACMAMI

Exemplo B.10: Aplicação do operador de mutação ACMaMi (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:maxQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:maxQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:minQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:minQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.11 OPERADOR DE MUTAÇÃO AXIOMCHANGEMINTOEXACTLY - ACMIEX

Exemplo B.11: Aplicação do operador de mutação ACMiEx (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:minQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:minQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:qualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:qualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.12 OPERADOR DE MUTAÇÃO AXIOMCHANGEMAXTOEXACTLY - ACMAEX

Exemplo B.12: Aplicação do operador de mutação ACMaEx (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:maxQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:maxQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:qualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:qualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.13 OPERADOR DE MUTAÇÃO AXIOMCHANGEEXACTLYTOMIN - ACEMI

Exemplo B.13: Aplicação do operador de mutação ACEMi (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:qualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:qualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:minQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:minQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.14 OPERADOR DE MUTAÇÃO AXIOMCHANGEEXACTLYTOMAX - ACEMA

Exemplo B.14: Aplicação do operador de mutação ACEMa (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:qualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:qualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:maxQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
        </owl:maxQualifiedCardinality>
        <owl:onClass rdf:resource=''Paciente'' />
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.15 OPERADOR DE MUTAÇÃO MAXCARDAXIOMUNDEF - MACAU

Exemplo B.15: Aplicação do operador de mutação MaCAU (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Atendimento''>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:maxQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
          </owl:maxQualifiedCardinality>
          <owl:onClass rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Atendimento''>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

B.16 OPERADOR DE MUTAÇÃO MINCARDAXIOMUNDEF - MICAU

Exemplo B.16: Aplicação do operador de mutação MiCAU (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Atendimento''>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:minQualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
          </owl:minQualifiedCardinality>
          <owl:onClass rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType=''Collection''>
          <rdf:Description rdf:about=''Atendimento'' />
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
</owl:Class>
```

B.17 OPERADOR DE MUTAÇÃO EXACARDAXIOMUNDEF - ECAU

Exemplo B.17: Aplicação do operador de mutação ECAU (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Atendimento''>
 <owl:equivalentClass>
   <owl:Class>
     <owl:intersectionOf rdf:parseType=''Collection''>
       <rdf:Description rdf:about=''Atendimento'' />
       <owl:Restriction>
         <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
         <owl:qualifiedCardinality rdf:datatype=''nonNegativeInteger''>1
         </owl:qualifiedCardinality>
         <owl:onClass rdf:resource=''Paciente'' />
       </owl:Restriction>
     </owl:intersectionOf>
   </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Atendimento''>
 <owl:equivalentClass>
   <owl:Class>
     <owl:intersectionOf rdf:parseType=''Collection''>
       <rdf:Description rdf:about=''Atendimento'' />
     </owl:intersectionOf>
   </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

B.18 OPERADOR DE MUTAÇÃO UNIVERSALAXIOMUNDEF - UAU

Exemplo B.18: Aplicação do operador de mutação UAU (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Atendimento''>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:someValuesFrom rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:allValuesFrom rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Atendimento''>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:someValuesFrom rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

B.19 OPERADOR DE MUTAÇÃO EXISTAXIOMUNDEF - EAU

Exemplo B.19: Aplicação do operador de mutação EAU (autor, 2019)

- ONTOLOGIA ORIGINAL

```
<owl:Class rdf:about=''Atendimento''>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:someValuesFrom rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:allValuesFrom rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

- ONTOLOGIA MUTANTE

```
<owl:Class rdf:about=''Atendimento''>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Atendimento'' />
        <owl:Restriction>
          <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
          <owl:allValuesFrom rdf:resource=''Paciente'' />
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

B.20 OPERADOR DE MUTAÇÃO AXIOMCHANGETRUETOFALSE - ACTF

Exemplo B.20: Aplicação do operador de mutação ACTF (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:hasValue rdf:datatype=''boolean''>true</owl:hasValue>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:hasValue rdf:datatype=''boolean''>false</owl:hasValue>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.21 OPERADOR DE MUTAÇÃO AXIOMCHANGEFALSETOTRUE - ACFT

Exemplo B.21: Aplicação do operador de mutação ACFT (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:hasValue rdf:datatype=''boolean''>false</owl:hasValue>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Atendimento''>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource=''temPessoaEnvolvida'' />
        <owl:hasValue rdf:datatype=''boolean''>true</owl:hasValue>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
```

B.22 OPERADOR DE MUTAÇÃO CHANGEINDIVIDUALTOSUBCLASS - CIS

Exemplo B.22: Aplicação do operador de mutação CIS (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:NamedIndividual rdf:about=''Homem'' >
    <rdf:type rdf:resource=''Pessoas'' />
  </owl:NamedIndividual>

- ONTOLOGIA MUTANTE
  <owl:Class rdf:about=''Homem''>
    <rdfs:subClassOf rdf:resource=''Pessoas'' />
  </owl:Class>
```

B.23 OPERADOR DE MUTAÇÃO INDIVIDUALDISJOINTUNDEF - IDU

Exemplo B.23: Aplicação do operador de mutação IDU (autor, 2019)

```
- ONTOLOGIA ORIGINAL
<rdf:Description>
    <rdf:type rdf:resource=''AllDifferent'' />
    <owl:distinctMembers rdf:parseType=''Collection''>
        <rdf:Description rdf:about=''Homem'' />
        <rdf:Description rdf:about=''Mulher'' />
    </owl:distinctMembers>
</rdf:Description>

- ONTOLOGIA MUTANTE
<rdf:Description>
    <rdf:type rdf:resource=''AllDifferent'' />
    <owl:distinctMembers rdf:parseType=''Collection''>
    </owl:distinctMembers>
</rdf:Description>
```

B.24 OPERADOR DE MUTAÇÃO INDIVIDUALSAMEUNDEF - ISU

Exemplo B.24: Aplicação do operador de mutação ISU (autor, 2019)

```
- ONTOLOGIA ORIGINAL
  <owl:NamedIndividual rdf:about=''Adulto''>
    <owl:sameAs rdf:resource=''Idoso'' />
  </owl:NamedIndividual>

- ONTOLOGIA MUTANTE
  <owl:NamedIndividual rdf:about=''Adulto'' />
  <owl:NamedIndividual rdf:about=''Idoso'' />
```

APÊNDICE C – ONTOLOGIAS OWL DO EXPERIMENTO I

C.1 ONTOLOGIA TIPOS DE ESPORTES - OTE

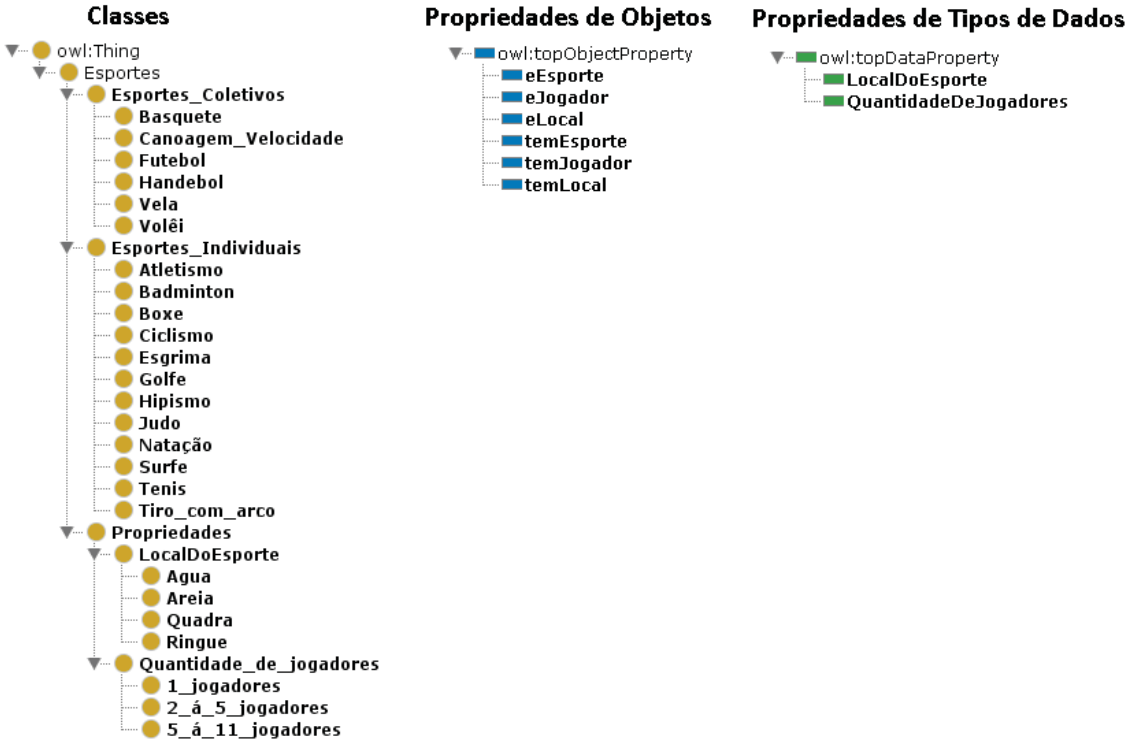


Figura C.1: Ontologia Tipos de Esporte (autor, 2019)s

C.2 ONTOLOGIA TIPOS DE ANIMAIS - OTA

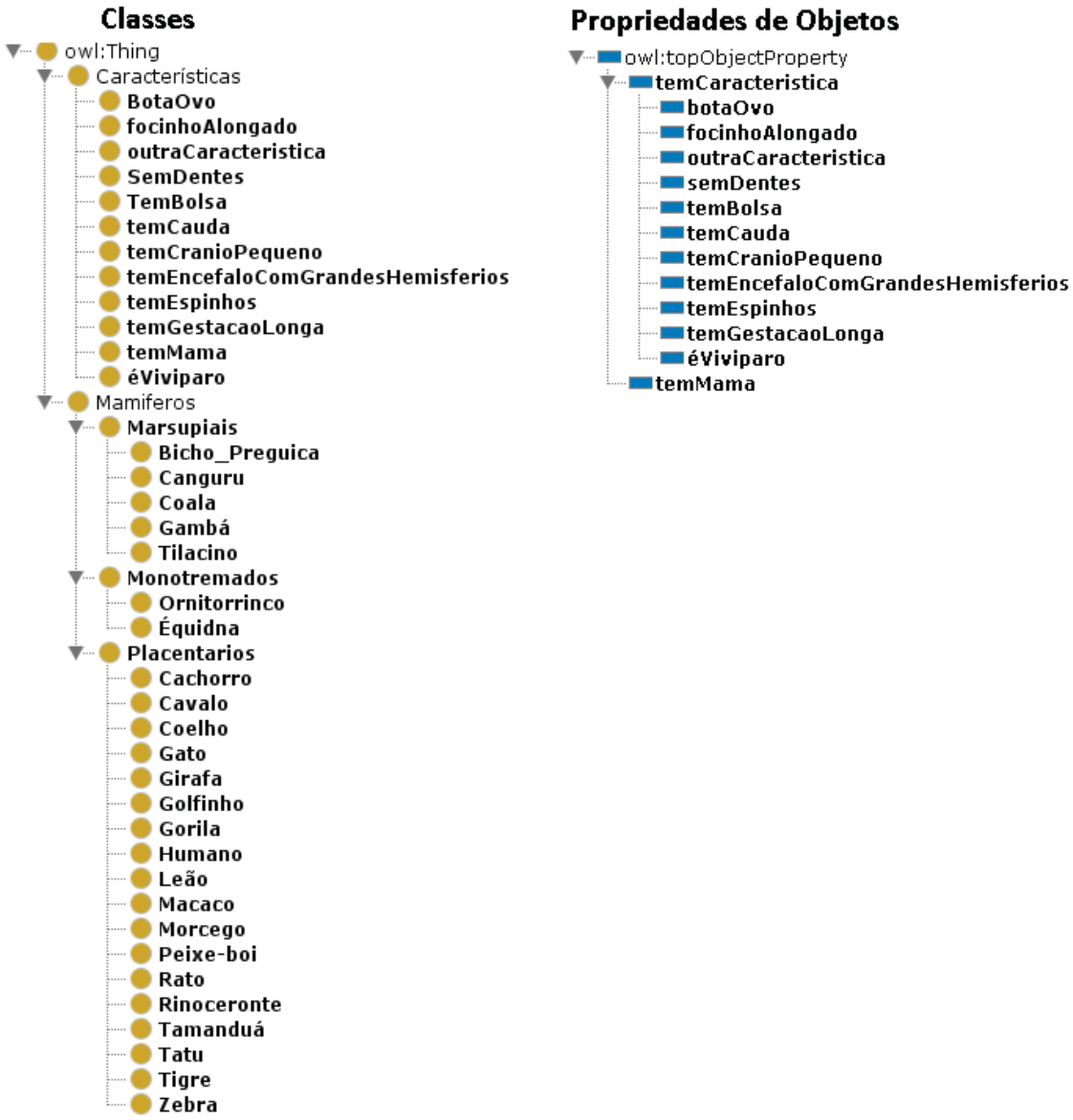


Figura C.2: Ontologia Tipos de Animais (autor, 2019)

C.3 ONTOLOGIA CONTINENTES E PAÍSES - OCP

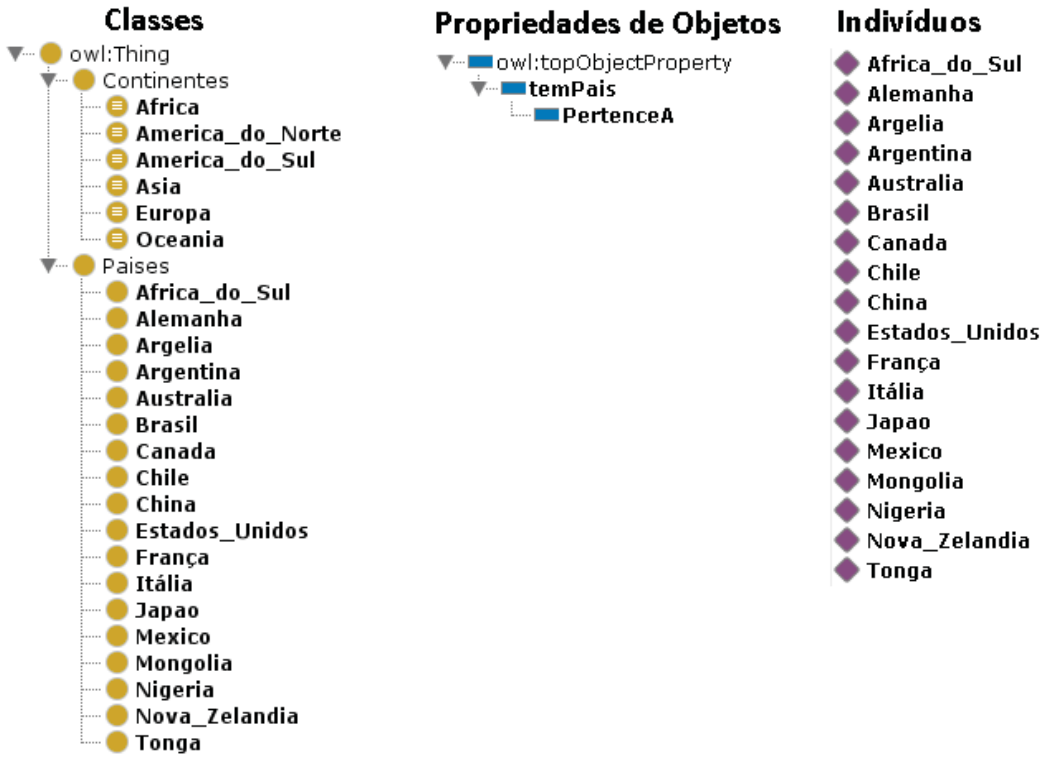


Figura C.3: Ontologia Continentes e Países (autor, 2019)

C.4 ONTOLOGIA CLASSIFICAÇÃO DE ESPORTES - OCE

Figura C.4: Ontologia Classificação de Esportes (autor, 2019)

C.5 ONTOLOGIA TIMES DE FUTEBOL AMERICANO - OTFA

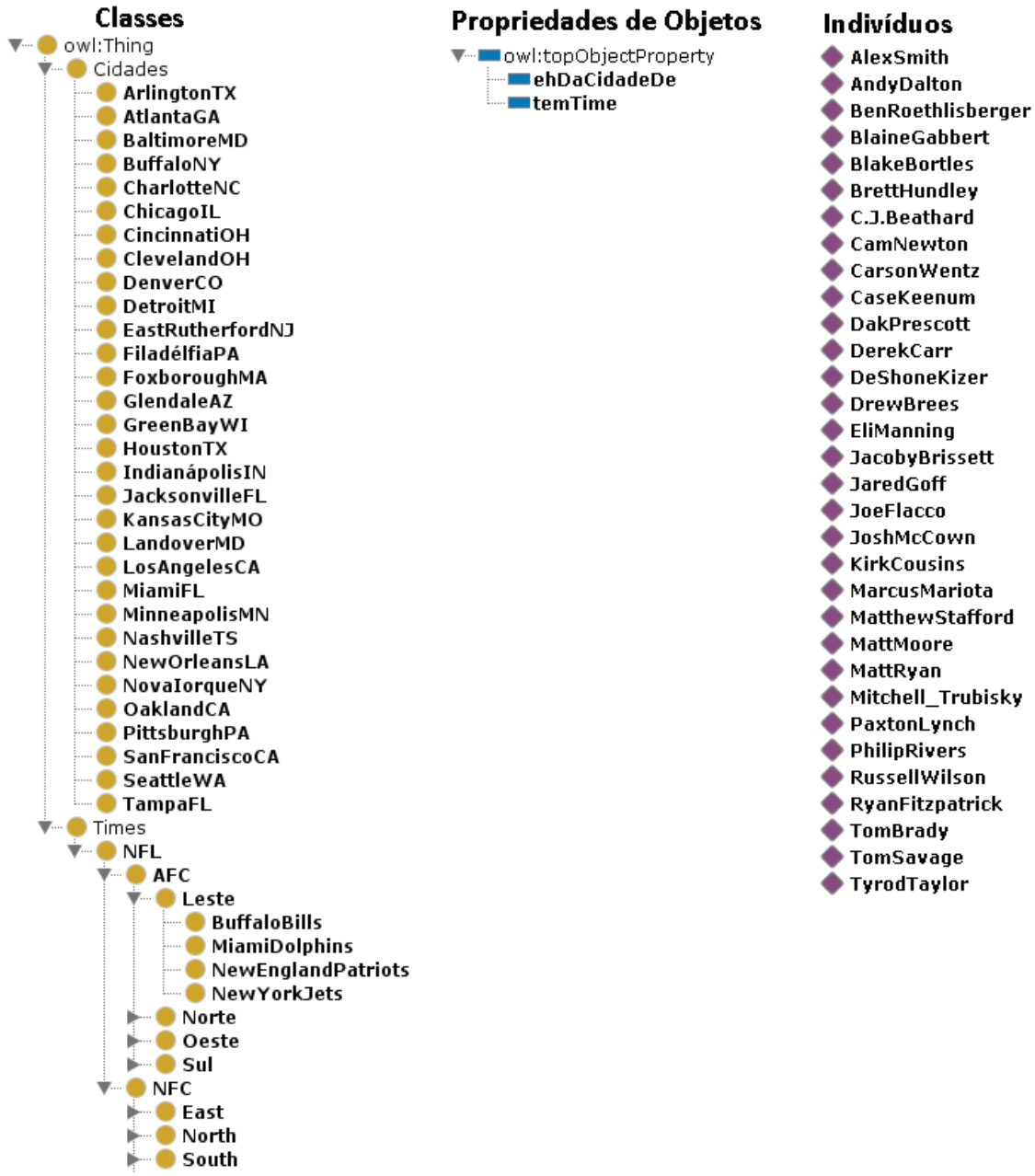


Figura C.5: Ontologia Times de Futebol Americano (autor, 2019)

C.6 ONTOLOGIA PONTOS TURÍSTICOS - OPT

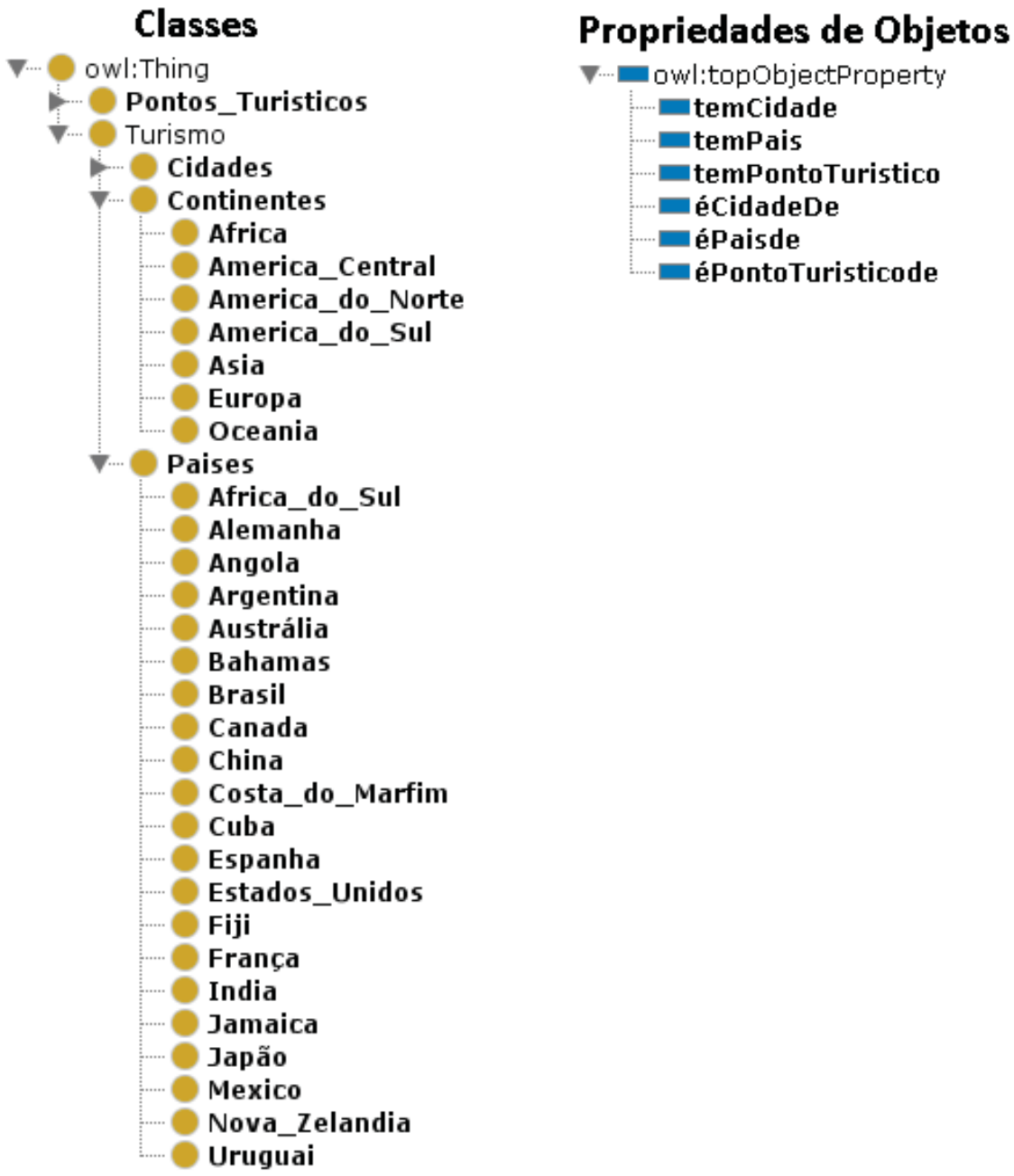


Figura C.6: Ontologia Pontos Turísticos (autor, 2019)

APÊNDICE D – ONTOLOGIAS OWL DO EXPERIMENTO II

D.1 ONTOLOGIA COMPARATIVA DE ANÁLISE DE DADOS - CDAO

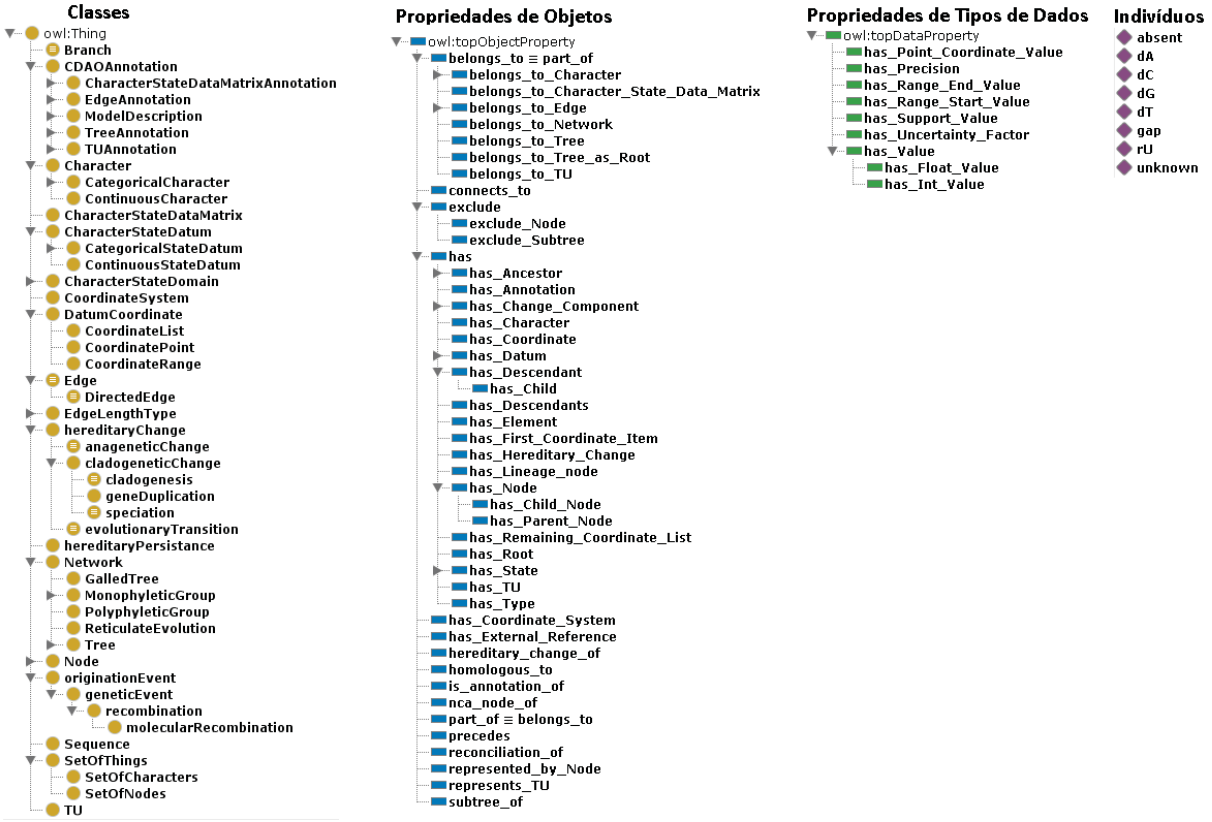


Figura D.1: Ontologia Comparativa de Análise de Dados (autor, 2019)

D.2 ONTOLOGIA DOENÇAS INFECCIOSAS - IDO

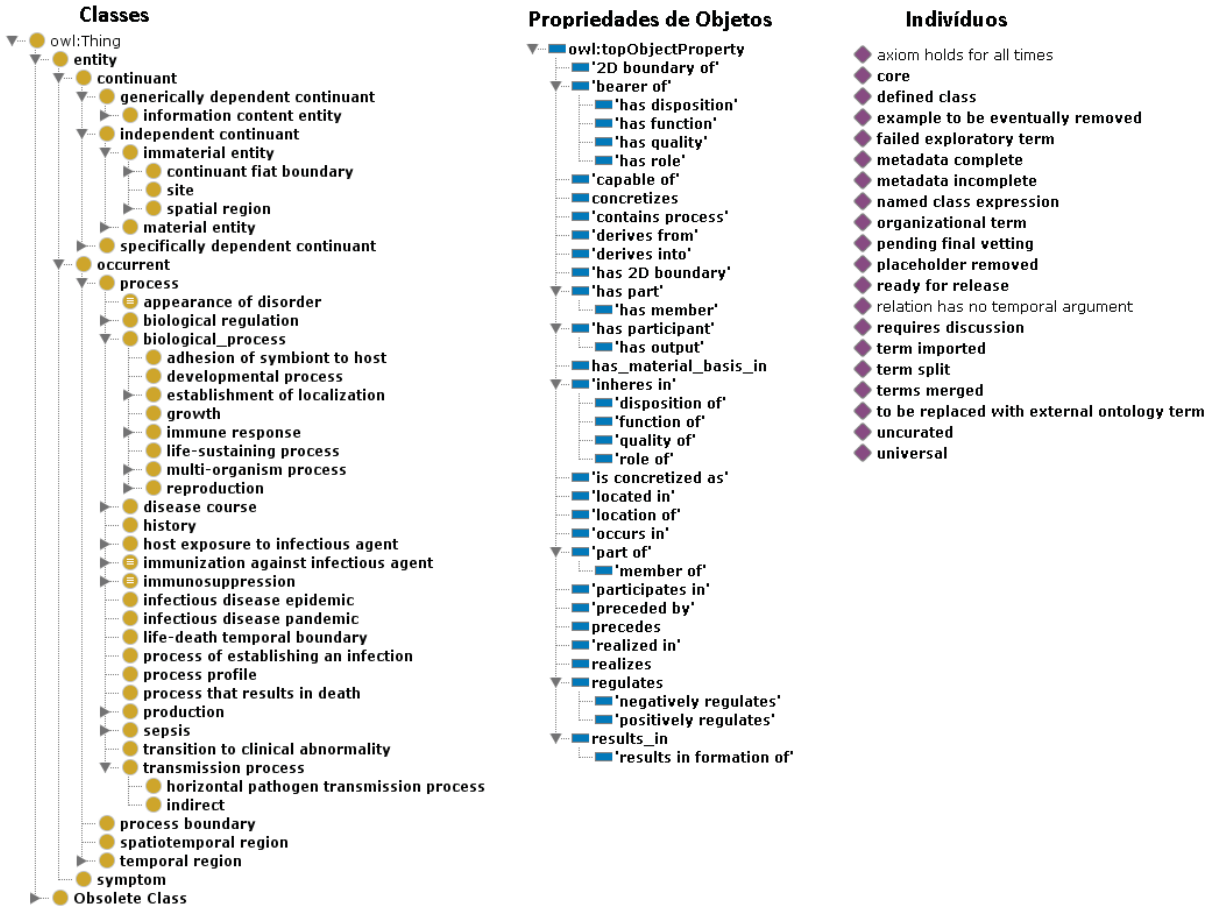


Figura D.2: Ontologia Doenças Infecciosas (autor, 2019)

D.3 ONTOLOGIA OBSTÉTRICA E NEONATAL - ONTONEO

Figura D.3: Ontologia Obstétrica e Neonatal (autor, 2019)

D.4 ONTOLOGIA ESTRUTURAS ORGANIZACIONAIS DE CENTROS E SISTEMAS DE TRAUMA - OOSTT

Figura D.4: Ontologia Estruturas Organizacionais de Centros e Sistemas de Trauma (autor, 2019)

D.5 ONTOLOGIA CICLO DE VIDA DE PARASITAS - OPL

Figura D.5: Ontologia Ciclo de Vida de Parasitas (autor, 2019)

D.6 ONTOLOGIA PRESCRIÇÃO DE MEDICAMENTOS - PDRO

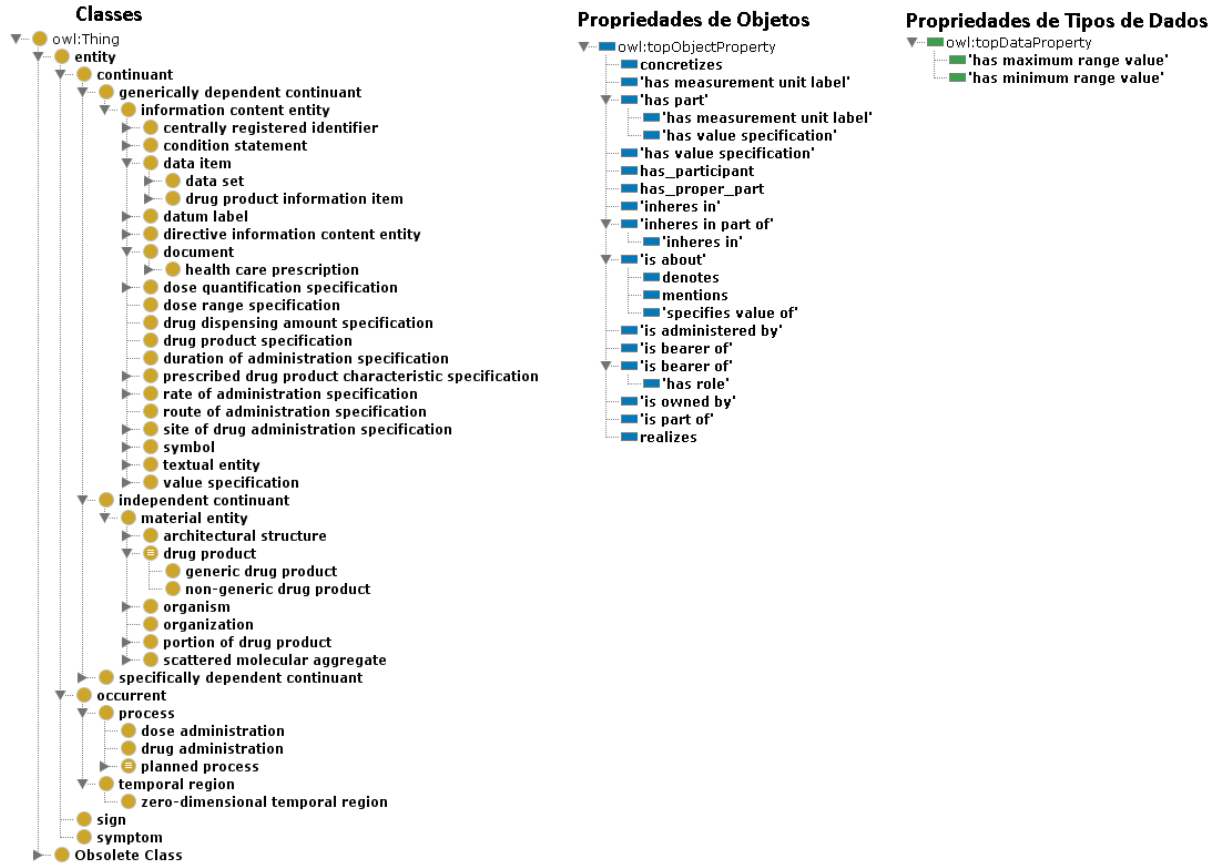


Figura D.6: Ontologia Prescrição de Medicamentos (autor, 2019)