ALEX MATEUS PORN

TESTE DE MUTAÇÃO PARA ONTOLOGIAS OWL

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profª. Drª. Leticia Mara Peres

CURITIBA

2014

ALEX MATEUS PORN

TESTE DE MUTAÇÃO PARA ONTOLOGIAS OWL

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profª. Drª. Leticia Mara Peres

CURITIBA

2014

Porn, Alex Mateus
Teste de mutação para ontologias OWL / Alex Mateus Porn . –
Curitiba, 2014
112 f. : il.; tabs.

Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientadora: Leticia Mara Peres

1. Ontologia. 2. Modelagem de dados. I. Peres, Leticia Mara. II.Título

CDD: 006.35

ALEX MATEUS PORN

TESTE DE MUTAÇÃO PARA ONTOLOGIAS OWL

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comissão formada pelos professores:

Orientadora: Profª. Drª. Leticia Mara Peres
Departamento de Informática, UFPR

Profª. Drª. Luciana Tricai Cavalini
Departamento de Informática, UFF

Prof. Dr. Marcos Didonet Del Fabro
Departamento de Informática, UFPR

Curitiba, 15 de Julho de 2014

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Alex Mateus Porn, avaliamos o trabalho intitulado, *"Teste de Mutação para Ontologias OWL"*, cuja defesa foi realizada no dia 15 de julho de 2014, às 14:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela: (X) **aprovação** do candidato. ( ) **reprovação** do candidato.

Curitiba, 15 de julho de 2014.

Profa. Dra. Leticia Mara Peres
DINF/UFPR – Orientadora

Profa. Dra. Luciana Tricai Cavalini
UFF – Membro Externo

Prof. Dr. Marcos Didonet Del Fabro
DINF/UFPR – Membro Interno

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

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

ATA DA DEFESA DE DISSERTAÇÃO DE MESTRADO EM INFORMÁTICA ALUNO: ALEX MATEUS PORN

No dia 15 de julho do ano de dois mil e quatorze, às 14:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná, foi realizada a sessão pública da defesa de Dissertação de Mestrado em Informática do aluno Alex Mateus Porn. Estavam presentes, além do candidato, os Membros da Comissão Examinadora composta pelos Professores Leticia Mara Peres(Orientadora), Luciana Tricai Cavalini e Marcos Didonet Del Fabro. Após a apresentação do trabalho do candidato, intitulado *"Teste de Mutação para Ontologias OWL"*, o mesmo foi arguido pela Comissão. A seguir, a Comissão reuniu-se em local reservado e decidiu, por unanimidade, pela aprovação do candidato, condicionada as alterações sugeridas pela mesma. O resultado foi então comunicado ao candidato e aos presentes na sessão pública. A seguir, o Presidente declarou encerrada a sessão da qual eu, Jucélia Miecznikowski, Secretária da Pós-graduação em Informática, lavrei a presente Ata, que depois de aprovada será assinada por mim, pelo Presidente, e pelos demais membros da Comissão.

Profa. Dra. Leticia Mara Peres
DINF/UFPR – Orientadora

Profa. Dra. Luciana Tricai Cavalini
UFF – Membro Externo

Prof. Dr. Marcos Didonet Del Fabro
DINF/UFPR – Membro Interno

Jucélia Miecznikowski
Secretária da PPGInf

AGRADECIMENTOS

Agradeço primeiramente a Deus pela motivação, saúde e sabedoria concedida para a conclusão desta pesquisa.

Agradeço a minha esposa Cibelli Cristina Porn, pela compreensão durante os dias de ausência. Agradeço muito, pois mais do que esposa, uma verdadeira amiga, foi uma peça fundamental durante esta etapa, incentivando e apoiando para a realização deste sonho.

Agradeço aos meus pais Lucas Porn e Edith Luisa Porn pelo reconhecimento e por sempre apoiarem e acreditarem no meu potencial.

Agradeço aos meus irmãos e demais familiares pela valorização e por sempre me apoiarem e torcerem para que o sucesso pudesse ser alcançado.

Agradeço principalmente a Profª. Drª. Leticia Mara Peres que me orientou com total dedicação, pela oportunidade e confiança para a conclusão deste estudo.

Agradeço ao Prof. Dr. Marcos Didonet Del Fabro, pelos conselhos e contribuições que positivamente enriqueceram este trabalho.

Agradeço a Profª. Drª. Luciana Tricai Cavalini pelas aulas lecionadas que possibilitaram conhecer um pouco mais de seu trabalho, contribuindo positivamente para o avanço e conclusão desta pesquisa.

Agradeço ao colega Hegler Correa Tissot por contribuir com o desenvolvimento desta pesquisa e, do mesmo modo, me apresentar ao mundo das ontologias.

Agradeço aos colegas de laboratório Vinicius Camargo Andrade, Rafael dos Passos Canteri, Luis Renato dos Santos e Carlos Eduardo Andrade Iatskiu, pelas contribuições concedidas e pelos momentos de distrações.

Agradeço a Profª. Mª. Edna Satiko Eiri Trebien, pelo incentivo, valorização e reconhecimento, e pelas incessantes substituições para a conclusão desta etapa.

Agradeço a Profª. Mª. Maria Genoveva Bordignon Esteves, a Profª. Drª. Simone dos Santos Junges e a Profª. Drª. Kelen dos Santos Junges, pelo incentivo e indicações para o ingresso no mestrado.

Agradeço ao Centro Universitário de União da Vitória - UNIUV, representado por sua reitoria, pela oportunidade concedida para a obtenção deste título. Também agradeço aos professores do colegiado de Sistemas de Informação pelas contribuições e apoio concedidos.

Agradeço a Universidade Federal do Paraná - UFPR, especialmente ao Programa de Pós-Graduação em Informática - PPGInf, pelo excelente curso oferecido.

Agradeço também a todos os amigos que direta ou indiretamente sempre apoiaram e torceram positivamente para que o objetivo fosse alcançado.

"A educação é a arma mais poderosa

que temos para mudar o mundo."

(Nelson Mandela)

RESUMO

Ontologias representam especificações formais sobre conceitos de um domínio e as relações entre eles. Elas desempenham um papel fundamental para descrever semânticas de dados e atuam como uma espinha dorsal em sistemas baseados em conhecimentos. A diversidade de métodos de modelagem de ontologias proporciona que defeitos sejam inseridos durante o desenvolvimento, possibilitando a ocorrência de falhas sintáticas e semânticas. Métodos de testes já foram propostos visando revelar falhas em ontologias, entretanto não possibilitam identificar os defeitos existentes. Com a finalidade de diminuir o número de defeitos na modelagem de ontologias, é apresentada neste trabalho a definição de um processo para a aplicação do teste de mutação para ontologias OWL, visando revelar defeitos em ontologias, assim como os seus tipos. Para a geração dos mutantes foram propostos 25 operadores de mutação, porém, neste trabalho 19 foram aplicados nos modelos em teste. Para a aplicação do teste foi realizado um estudo de caso no contexto de registros eletrônicos de saúde, onde arquétipos openEHR definidos na linguagem ADL, foram convertidos para ontologias OWL e submetidos ao processo de teste. Para auxiliar o processo de geração dos mutantes e execução dos dados de teste, utilizou-se a ferramenta *Protégé* de criação e edição de ontologias. O método de teste mostrou-se eficaz para determinar a corretude de ontologias, sendo possível gerar um alto número de mutantes e um alto escore de mutação através dos operadores propostos e dos dados de teste utilizados, podendo-se identificar e analisar os defeitos inseridos durante o desenvolvimento.

Palavras-Chave: Ontologias. Defeitos de modelagem. Teste de mutação.

ABSTRACT

Ontologies represent formal specifications about the domain concepts and the relation between them. They have a fundamental role to describe data semantics and act as a backbone in systems knowledge-based. The insertion of defects during an ontology development is common, causing the occurrence of syntactics and semantics faults. Test methods have been proposed aiming to show faults in ontologies. However these methods does not identify the faults sources. In order to decrease the number of these defects in an ontology design, the definition of a process for the application of mutation test to OWL ontologies is presented. It is possible to show existing defects and their types. To generate the mutants, 25 mutation operators have been proposed, but on this paper 19 were applied on models being tested. The application of the test is done by a case study. It was conducted on the context of electronic health records, with openEHR archetypes. These archetypes were written in ADL language, converted to OWL ontologies and submitted to the testing process. It was used Protégé, a tool of creation and edition of ontologies, to assist the process of mutants generation and test data execution. The test method was efficacious to determine the correctness of ontologies and was possible to generate a sufficient number of mutants and a high mutation score. Through the proposed operators and the test data used, it was possible identify and analyze the defects inserted during the development.

Key-words: Ontologies. Design defects. Mutation test.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ADL	-	Archetype Definition Language
AM	-	Archetype Model
AOM	-	Archetype Object Model
CID10	-	Código Internacional de Doenças
CKM	-	Clinical Knowledge Manager
CQ	-	Competency Question
LOINC	-	Logical Observation Identifiers Names and Codes
RDFS	-	Resource Description Framework Schema
RES	-	Registro Eletrônico de Saúde
RM	-	Reference Model
OWL	-	Ontology Web Language
OWL-S	-	Ontology Web Language for Services
UML	-	Unified Modeling Language
W3C	-	World Wide Web Consortium

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

O termo ontologia vem do campo da filosofia que lida com a natureza e a organização do ser ou existência. Segundo o dicionário Aurélio [17], é uma "ciência do ser em geral, que considera o ser em si mesmo, independente do modo pelo qual se manifesta". Deste modo, com o auxílio das ontologias, filósofos tentam responder a questões que definem o que é um ser e quais são as características comuns de todos os seres [48]. Neste contexto, o termo "ser" é entendido como tendo uma única representação.

Esse termo foi adotado também pelas comunidades de inteligência artificial e gestão do conhecimento, para se referir a conceitos e termos que podem ser usados para descrever algum domínio, ou construir uma representação desse. Neste caso, ontologia refere-se a um termo técnico denotando um artefato, que é projetado para a finalidade de permitir a modelagem de conhecimento sobre algum domínio, real ou imaginário [34, 35].

As ontologias demonstraram ser úteis para apoiar a especificação e desenvolvimento de qualquer sistema de computação, principalmente devido a algumas vantagens na sua utilização: fornecem uma descrição exata e um vocabulário para representação do conhecimento, assim como o seu compartilhamento; possibilitam o mapeamento da linguagem da ontologia sem que seja alterada a sua conceitualização; e estendem o uso de uma ontologia genérica para um domínio específico [35].

Na computação, uma ontologia é definida com base em um conjunto de conceitos com os quais modela um domínio de conhecimento específico. Estes conceitos são tipicamente classes, propriedades e relacionamentos entre os membros das classes. As definições destes conceitos incluem informações sobre os significados e as restrições sobre sua coerência na lógica aplicada [34].

A diversidade de representações de conceitos e características na modelagem das ontologias, possibilita a descrição de diversos tipos de domínios de conhecimento específico,

de modo que já foram utilizadas em várias áreas do conhecimento como geoprocessamento [19], educação [14] e saúde [56], destacando-se nesta área com uma possibilidade flexível de representação de informações clínicas. Esta variedade de representações facilita a ocorrência de defeitos sintáticos ou semânticos no desenvolvimento, como a aplicação incorreta de operadores e qualificadores de restrições, que definem a instanciação de classes ou indivíduos da ontologia, ou ao não defini-los, como não representar a disjunção ou equivalência entre classes, ou também, ao definir incorretamente domínios e limites de classes e propriedades.

A existência de defeitos [32, 62, 58, 13] nas ontologias pode causar falhas como a instanciação incorreta de classes e objetos, retornando resultados incompletos ou incorretos aos usuários, podendo também invalidar uma ontologia inteira. Estes defeitos podem não ser detectados durante a execução da ontologia e geralmente são cometidos na definição e representação de axiomas, que são definições utilizadas pelo modelador para representar os conceitos do domínio na ontologia.

Métodos de testes já foram propostos para verificar e validar ontologias [29, 68], como analisar a instanciação de classes e indivíduos ou verificar inconsistências ao adicionar novos objetos na ontologia, ambos realizados durante a execução das ontologias com auxílio de um classificador (do inglês, *reasoner*). No trabalho de Lee, Bai e Chen [46], é aplicado o teste de mutação para testes de *Web Services* baseados em ontologias do tipo OWL-S (do inglês, *Ontology Web Language for Services*) que é uma especificação para escrever serviços web [47]. Estes tipos de testes, exceto o trabalho de Lee, Bai e Chen [46], possibilitam analisar os resultados obtidos após a execução de uma ontologia, mas não permitem revelar quais são os defeitos existentes.

É necessária a definição de uma técnica de teste para ontologias, que permita determinar quais são os defeitos existentes no momento em que são revelados. Deste modo, as ontologias necessitam ser testadas dentro do contexto de conhecimento ao qual se propõem a descrever, assim como um desenvolvedor de software promove os testes necessários para que o programa atenda às necessidades ao qual ele foi criado.

O critério de teste de mutação para programas é utilizado neste trabalho para testar

ontologias OWL (do inglês, *Ontology Web Language*) devido as suas características, onde possíveis defeitos cometidos no desenvolvimento de um software são revelados a partir de erros inseridos no código fonte do aplicativo, de acordo com operadores de mutação que definem estas modificações. Assim, é proposta nesta dissertação de mestrado a aplicação deste critério para testar ontologias OWL. Para isto, são definidos operadores de mutação para ontologias OWL, com base nos defeitos típicos que podem ser cometidos durante o desenvolvimento [32, 62, 58, 13] e nas características básicas da linguagem OWL [1, 2].

Cada operador de mutação proposto é utilizado para gerar o maior número possível de ontologias com o defeito que o operador se propõe a aplicar, sendo estas ontologias definidas como ontologias mutantes. Os casos de testes são definidos com base em consultas escritas em linguagens de ontologias, como *DL Query* [60], sendo executados com auxílio da ferramenta *Protégé* [67], de criação e edição de ontologias, de modo que esta ferramenta também é utilizada no processo de geração das ontologias e comparação dos resultados obtidos.

Pretende-se com a aplicação desta técnica, validar o teste de mutação como uma técnica eficaz para o teste de ontologias, definir um escore de mutação a fim de avaliar a adequação dos operadores de mutação propostos, revelando o maior número de defeitos possíveis que não seriam revelados sem a aplicação desta técnica.

1.1 Motivação

Dado o contexto acima, têm-se as seguintes questões como motivação para o presente trabalho:

- Ontologias OWL podem conter defeitos de acordo com a sua especificação, de modo que esses defeitos podem não ser detectados por classificadores;
- Técnicas de validação de ontologias têm sido estudadas e definidas para o teste de ontologias e são fundamentais para assegurar a qualidade da modelagem;
- O teste de mutação tem mostrado-se um dos mais eficazes em revelar defeitos em software, podendo ser adaptado para o conceito de ontologias.

1.2 Objetivo geral

Desenvolver uma técnica de validação para diminuir o número de erros na modelagem de ontologias. Para isto o teste de mutação em programas é adaptado ao conceito de ontologias OWL.

1.2.1 Objetivos específicos

- Estudar e tipificar os principais defeitos inseridos durante o desenvolvimento de ontologias OWL;
- Definir operadores de mutação baseados nos trabalhos de Lee, Bai e Chen[46] e Derezinska [23], que propõem operadores para sistemas orientados a objetos e *Web Services* baseados em ontologias OWL-S;
- Estabelecer um processo de aplicação do teste de mutação para a geração das ontologias mutantes e dados de teste, assim como para a execução das mesmas e comparação dos resultados com a ontologia original.

1.3 Organização do trabalho

Este trabalho está subdividido em seis capítulos. No Capítulo 2 é apresentada a definição e representação de ontologias OWL e os principais erros de modelagem cometidos pelos desenvolvedores. No Capítulo 3 é descrito os conceitos de teste de software e o teste de mutação, assim como alguns métodos de testes utilizados em ontologias.

No Capítulo 4 é apresentada a aplicação do teste de mutação para ontologias OWL, sendo definidos os operadores de mutação, assim como os passos necessários para a execução do teste. No Capítulo 5 é apresentado um estudo de caso realizado no contexto de registros eletrônicos de saúde, onde conceitos do domínio clínico representados em arquétipos openEHR são convertidos para ontologias OWL, para em seguida serem aplicadas ao teste de mutação. No Capítulo 6 são apresentadas as conclusões e sugestões de trabalhos futuros para a continuação deste estudo.

CAPÍTULO 2

ONTOLOGIAS

Uma ontologia define os termos usados para descrever e representar um domínio específico do conhecimento. É uma descrição de conceitos e relacionamentos que pode ser usada por pessoas ou agentes de software para compartilhar informações dentro deste domínio. Assim, pode ser usada como uma estrutura unificadora para representação semântica de informação [34, 16].

Ontologias constituem-se em definições de conceitos, classes, propriedades, relações, restrições e axiomas sobre um determinado domínio do conhecimento, e padronizam significados através de identificadores semânticos, os quais podem representar o mundo real e conceitual [34, 28]. Para isso, requerem o uso de um vocabulário específico usado para descrever determinada realidade, sendo possível capturar os conceitos e relações em algum domínio de conhecimento, e um conjunto de axiomas permitem restringir a sua interpretação [16].

Um modelo de domínio representado por uma ontologia é definido através da representação de classes, instâncias das classes e relacionamentos entre as instâncias da ontologia. Sendo assim, as *classes* são 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 [9, 31].

A estruturação das *extensões de classe*, assim como da hierarquia das classes na ontologia, ocorre através da aplicação de um *reasoner*, neste trabalho denominado como *classificador*. Um classificador é um programa que analisa expressões lógicas em um conjunto de axiomas explícitamente definidos, fornecendo suporte automatizado para classificação, depuração e consultas sobre a ontologia [22].

Formalmente, uma ontologia O pode ser definida como uma quíntupla (C, I, P, V, A), onde [45, 70]:

- $C = (c_1, c_2, ..., c_n)$ é um conjunto de n classes, onde c_i é a i-ésima classe.
- $I = (I_1, I_2, ..., I_m)$, onde I_i é o conjunto de instâncias (indivíduos) da classe c_i.
- $P = (p_1(c_{1,1}, c_{1,2}), p_2(c_{2,1}, c_{2,2}), ..., p_m(c_{m,1}, c_{m,2}))$ 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 pela propriedade p_j.
- $V = (T_1, T_2, ..., T_m)$, 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.
- A é um conjunto de outros axiomas, como restrições que definem outras características da ontologia O, como equivalência de classes ou definição de classes disjuntas.

O potencial das ontologias encontra-se na habilidade de criar relacionamentos entre classes e instâncias, e atribuir propriedades para estes relacionamentos permitindo aplicar deduções sobre eles [41]. 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. Assim, o raciocínio lógico baseado na ontologia permite inferir que qualquer indivíduo de uma classe é considerado também como pertencente a todas as superclasses da qual sua classe deriva.

2.1 Ontologias OWL

A linguagem OWL é uma linguagem para definição e instanciação de ontologias Web, definida pela W3C (*World Wide Web Consortium*) [2]. Uma ontologia OWL especifica como derivar consequências lógicas através da semântica formal OWL, esclarecendo fatos que não estão presentes na ontologia mas são vinculados pela semântica [1].

As expressões lógicas de uma ontologia OWL são definidas pela representação de axiomas, dos quais modelam-se o domínio de conhecimento específico. Conforme Vrandecic et al. [69], axiomas descrevem relações entre elementos de uma ontologia, sejam eles classes, propriedades ou indivíduos. Assim, uma ontologia pode ser considerada como um conjunto de axiomas, que através dos construtores OWL que permitem a instanciação de objetos em classes e subclasses, dos quais aplicam-se em toda a estrutura das ontologias.

O metamodelo[1] de definição de ontologias ODM (do inglês, *Ontology Definition Metamodel*) apresenta dois modelos para a representação destas estruturas, sendo definidos pelas linguagens RDFS (do inglês, *Resource Description Framework Schema*) e OWL. O modelo RDFS é uma meta linguagem que além de representar seus conceitos, define também o modelo OWL [31]. Deste modo, OWL baseia-se no metamodelo RDFS, estendendo todos os seus conceitos. O metamodelo de ontologias OWL é apresentado na Figura 2.1.

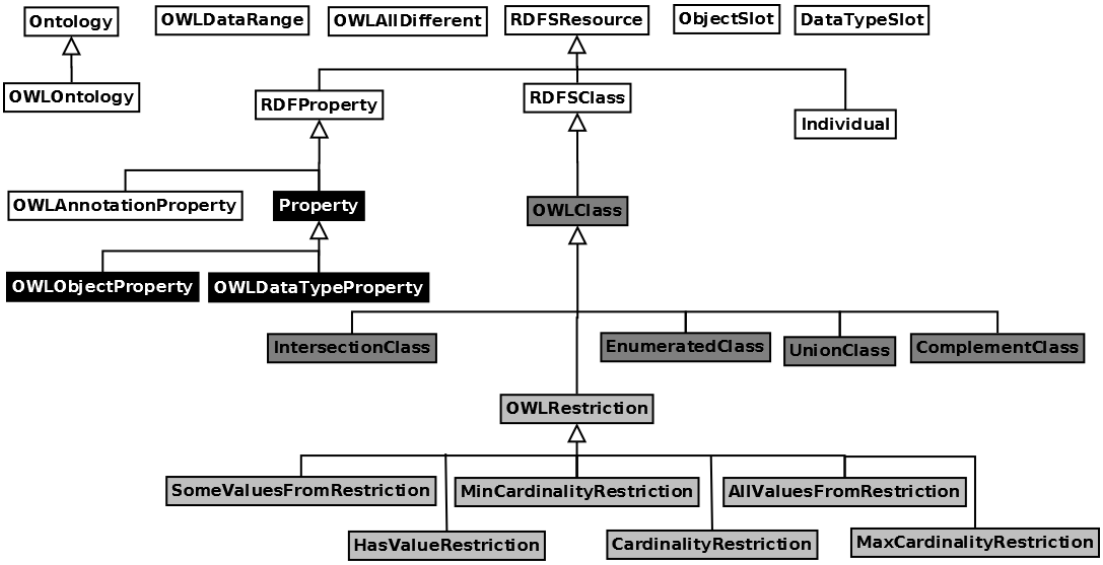

Figura 2.1: Metamodelo de Ontologias OWL

As classes, propriedades e restrições das ontologias OWL são apresentadas na sequência.

[1]Um metamodelo define possíveis estruturas e significados para os elementos de um modelo [5]

2.1.1 Classes

As classes de ontologias OWL podem ser construídas como complemento ou como uma união ou intersecção de outras classes, através dos operadores *owl:complementOf*, *owl:unionOf* e *owl:intersectionOf*, representados respectivamente no metamodelo pelas classes *ComplementClass*, *UnionClass* e *IntersectionClass*, conforme a Figura 2.1, destacado pelas caixas em cinza escuro. Do mesmo modo, também podem ser representadas através da definição de uma lista exaustiva de indivíduos que deverão ser instanciados à classe, através do operador *owl:oneOf*, representado pela classe *EnumeratedClass* no metamodelo. Em contrapartida outros dois tipos de associações são definidas, *owl:equivalentClass* que determina que duas classes possuem a mesma extensão de classe e, *owl:disjointWith* que determina que duas extensões de classes não possuem quaisquer indivíduos comuns [31].

Os operadores *owl:intersectionOf*, *owl:unionOf* e *owl:complementOf*, podem ser visualizados como os operadores "AND", "OR" e "NOT" utilizados em lógica descritiva.

O operador *owl:intersectionOf* declara uma classe de indivíduos comuns em todas as classes descritas em uma lista, sendo análogo à conjunção lógica *AND* [18].

O Exemplo 1 representa o operador *owl:intersectionOf* descrevendo uma classe instanciada com o indivíduo *Paraná*, pois é o único indivíduo comum em ambas as classes.

Exemplo 1:

```
<owl:Classs >
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class>
            <owl:oneOf rdf:parseType="Collection">
                <owl:Thing rdf:about="#Paraná" />
                <owl:Thing rdf:about="#Amazonas" />
            </owl:oneOf>
        </owl:Class>
        <owl:Class>
            <owl:oneOf rdf:parseType="Collection">
                <owl:Thing rdf:about="#Bahia" />
                <owl:Thing rdf:about="#Paraná" />
            </owl:oneOf>
        </owl:Class>
    </owl:intersectionOf>
</owl:Class>
```

O operador *owl:unionOf* declara uma classe, cuja extensão de classe são indivíduos instânciados em pelo menos uma das classes descritas em uma lista. Este operador é análogo a disjunção lógica *OR* [18].

De acordo com o Exemplo 2, o operador *owl:unionOf* define uma classe cuja extensão de classe refere-se aos indivíduos *Paraná, Amazonas e Bahia*, devido os mesmos serem instâncias de no mínimo uma das classes listadas.

Exemplo 2:

```
<owl:Classs >
    <owl:unionOf rdf:parseType="Collection">
        <owl:Class>
            <owl:oneOf rdf:parseType="Collection">
                <owl:Thing rdf:about="#Paraná" />
                <owl:Thing rdf:about="#Amazonas" />
            </owl:oneOf>
        </owl:Class>
        <owl:Class>
            <owl:oneOf rdf:parseType="Collection">
                <owl:Thing rdf:about="#Bahia" />
                <owl:Thing rdf:about="#Paraná" />
            </owl:oneOf>
        </owl:Class>
    </owl:unionOf>
</owl:Class>
```

O operador *owl:complementOf* declara uma classe, cuja extensão são indivíduos que não pertencem à classe declarada. Este operador é análogo a negação lógica *NOT* [18].

O Exemplo 3 apresenta a definição de uma classe de todos os indivíduos que não pertencem a classe *Norte*, definida no operador *owl:complementOf*.

Exemplo 3:

```
<owl:Class>
    <owl:complementOf>
        <owl:Class rdf:about="#Norte"/>
    </owl:complementOf>
</owl:Class>
```

O operador *owl:oneOf*, declara uma classe através da definição dos indivíduos que devem ser instanciados à esta classe [2].

De acordo com o Exemplo 4, o operador *owl:oneOf* define uma classe cujos indivíduos são listados pela sintaxe *rdf:about*, a qual refere-se a instanciação de um indivíduo específico [9].

Exemplo 4:

```
<owl:Class>
    <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#RioGrandeDoSul"/>
        <owl:Thing rdf:about="#SantaCatarina"/>
        <owl:Thing rdf:about="#Parana"/>
        <owl:Thing rdf:about="#SãoPaulo"/>
        <owl:Thing rdf:about="#RioDeJaneiro"/>
        <owl:Thing rdf:about="#EspiritoSanto"/>
    </owl:oneOf>
</owl:Class>
```

Com base nas classes definidas, OWL apresenta três construtores para a representação de classes através de axiomas. O construtor *rdfs:subClassOf* permite definir uma hierarquia de classes através da declaração de que uma classe é subclasse de outra [2]. Se uma classe C_1 é definida como subclasse de uma classe C_2, então o conjunto de indivíduos da classe C_1 deve ser também um conjunto de indivíduos da classe C_2. Conforme Bechhofer et al. [9], uma classe é por definição uma subclasse dela mesma.

No Exemplo 5 a classe *Paraná* é definida como uma subclasse da classe *Sul*, determinando que todos os indivíduos da classe *Paraná* também pertencem a classe *Sul*.

Exemplo 5:

```
<owl:Class rdf:ID="Paraná">
    <rdfs:subClassOf rdf:resource="#Sul" />
</owl:Class>
```

O construtor *owl:equivalentClass* associa uma classe a outra classe, determinando que as duas classes envolvidas possuem a mesma extensão de classe, ou seja, ambas as classes contém exatamente o mesmo conjunto de indivíduos [9].

Conforme demonstrado no Exemplo 6, ambas as classes *Paraná* e *PR* são definidas como equivalentes através do construtor *owl:equivalentClass*, determinando que ambas possuem o mesmo conjunto de indivíduos.

Exemplo 6:
<owl:Class rdf:about="#Paraná">
<equivalentClass rdf:resource="#PR"/>
</owl:Class>

O construtor *owl:disjointWith* permite apresentar que uma classe não tem membros em comum com outra classe, definindo-as como disjuntas [9].

De acordo com o Exemplo 7, todos os indivíduos que são associados à classe *Paraná*, em nenhuma hipótese podem ser definidos como instâncias da classe *SantaCatarina*, e o mesmo ocorre para os indivíduos associados à classe *SantaCatarina*, devido as duas classes serem definidas como disjuntas pelo construtor *owl:disjointWith*.

Exemplo 7:
<owl:Class rdf:about="#Paraná">
<owl:disjointWith rdf:resource="#SantaCatarina"/>
</owl:Class>

Nesta seção foram apresentados todos os operadores OWL para representações de classes definidos pela W3C e que são utilizados neste trabalho.

2.1.2 Propriedades

As propriedades em ontologias OWL são relações binárias entre indivíduos de uma ontologia ou entre indivíduos e valores de dados e permitem afirmar fatos gerais sobre os membros de classes e fatos específicos sobre indivíduos [1, 33]. Assim, é possível a inserção de dois tipos principais de propriedades, conforme representado pela Figura 2.1, destacado pelas caixas pretas: *Object Properties* (Propriedades de Objetos) e *Data Type Properties* (Propriedades de Tipos de Dados) [1]. Enquanto as *Object Properties* conectam um indivíduo a outro, por exemplo, um *Paciente* a um *Exame_realizado*, as *Data Type Properties* conectam um indivíduo a um determinado tipo de dado literal, por exemplo, um *Paciente* a um dado literal numérico inteiro que representa sua idade [31, 38].

Similarmente às classes, as propriedades em OWL são representadas através de diversos axiomas que definem as ontologias. As propriedades podem ser definidas em hierarquias através do operador *rdfs:subPropertyOf*, que determina que uma propriedade é uma

subpropriedade de outra. Formalmente definindo, significa que se P_1 é subpropriedade de P_2, então o conjunto extensão de propriedade de P_1, é também um subconjunto da extensão de propriedade de P_2 [9].

Conforme o Exemplo 8, a propriedade *hasMother* é uma subpropriedade da propriedade *hasParent*. Este tipo de operador pode ser aplicado tanto para as propriedades de objetos como de tipos de dados em OWL.

Exemplo 8:

```
<owl:ObjectProperty rdf:ID="hasMother">
    <rdfs:subPropertyOf rdf:resource="#hasParent"/>
</owl:ObjectProperty>
```

Para cada propriedade pode ser definido o operador *rdfs:domain*, representando que uma propriedade pertence ao domínio de uma ou mais classes respectivamente, de modo que um axioma pode ser definido utilizando-se múltiplos operadores deste tipo [9].

O Exemplo 9 define que o domínio da propriedade *hasSouthRegion* pode ser tanto um ou vários indivíduos da classe *Paraná* ou da classe *SantaCatarina*, ou seja, esta propriedade deve ser usada somente por indivíduos destas duas classes.

Exemplo 9:

```
<owl:ObjectProperty rdf:ID="hasSouthRegion">
    <rdfs:domain>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Paraná"/>
                <owl:Class rdf:about="#SantaCatatina"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:domain>
</owl:ObjectProperty>
```

Da mesma forma como o operador *rdfs:domain*, podem haver múltiplos operadores *rdfs:range* em uma propriedade, definindo que os valores de uma propriedade devem ser instâncias de uma ou mais classes respectivamente, ou valores de dados de um limite determinado [9].

Ao contrário do Exemplo 9, no Exemplo 10 os valores da propriedade *hasSouthRegion* devem pertencer à extensão de classe da classe *Paraná* ou da classe *SantaCatarina*.

Exemplo 10:

```
<owl:ObjectProperty rdf:ID="hasSouthRegion">
    <rdfs:range>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Paraná"/>
                <owl:Class rdf:about="#SantaCatatina"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:range>
</owl:ObjectProperty>
```

Conforme o domínio e limite definidos para as propriedades da ontologia, é possível a representação de relações entre as propriedades através dos operadores *owl:equivalentProperty* e *owl:inverseOf*, do mesmo modo como são realizadas entre indivíduos e classes.

O operador *owl:equivalentProperty* define que duas propriedades ou mais são equivalentes, ou seja, possuem o mesmo domínio e limite de outra propriedade [9, 18]. Ao contrário, o operador *owl:inverseOf* define uma relação inversa entre propriedades, onde um axioma na forma de *"P_1 owl:inverseOf P_2"* caracteriza que para cada par (x, y) da extensão de propriedade de P_1, existe um par (y, x) da extensão de propriedade de P_2, assim, *owl:inverseOf* é uma propriedade simétrica [9].

Com base no Exemplo 11, pode-se definir que o operador *owl:inverseOf* caracteriza que um indivíduo associado a propriedade *isChild*, possui como propriedade inversa a propriedade *hasParent*.

Exemplo 11:

```
<owl:ObjectProperty rdf:ID="isChild">
    <owl:inverseOf rdf:resource="#hasParent"/>
</owl:ObjectProperty>
```

De acordo com as possibilidades de cardinalidades aplicadas para a instanciação de classes nas ontologias OWL, também é possível a definição de restrições de cardinalidades

em propriedades através dos operadores *owl:FunctionalProperty* e *owl:InverseFunctional-Property*.

A definição de uma propriedade com o operador *owl:FunctionalProperty*, determina que pode haver somente um valor de y para cada instância de x, não podendo haver dois valores distintos y_1 e y_2, tal que os pares (x, y_1) e (x, y_2) sejam ambos instâncias desta propriedade. A definição desta característica é dita ter cardinalidade mínima 0 e cardinalidade máxima 1, de modo que ambos os dois tipos de propriedades podem ser declaradas como funcionais.

De acordo com o Exemplo 12, o operador *owl:FunctionalProperty* caracteriza que os indivíduos da classe *Man* que utilizam a propriedade *husband*, podem estar associados a no máximo um indivíduo da classse *Woman*.

Exemplo 12:
<owl:ObjectProperty rdf:ID="husband">
<rdf:type rdf:resource="&owl;FunctionalProperty" />
<rdfs:domain rdf:resource="#Woman" />
<rdfs:range rdf:resource="#Man" />
</owl:ObjectProperty>

Já ao definir uma propriedade com o operador *owl:InverseFunctionalProperty*, afirma-se que um valor de y pode somente ser um valor de P para uma única instância x, ou seja, não pode haver duas instâncias distintas x_1 e x_2, tal que ambos os pares (x_1, y) e (x_2, y) são instâcias de P [9].

Ao contrário do Exemplo 12, no Exemplo 13 um indivíduo da classe *Man* pode estar associado a no máximo um indivíduo da classe *Woman* que utiliza a propriedade *husband*.

Exemplo 13:
<owl:InverseFunctionalProperty rdf:ID="husband">
<rdfs:domain rdf:resource="#Woman"/>
<rdfs:range rdf:resource="#Man"/>
</owl:InverseFunctionalProperty>

Este construtor é utilizado somente para propriedades de objetos, devido o mesmo ser uma instância da classe *owl:InverseFunctionalProperty* que é definida como subclasse da classe OWL *owl:ObjectProperty*.

As propriedades OWL também possibilitam a definição de características lógicas através dos operadores *owl:TransitiveProperty* e *owl:SymmetricProperty*, de modo que ao representar uma propriedade com o operador *owl:TransitiveProperty*, significa que se um par (x, y) é uma instância de *P*, e outro par (y, z) também é uma instância de *P*, então infere-se que o par (x, z) também é uma instância de *P*. Da mesma forma como o operador de propriedades funcionais inversas, este tipo de operador também é aplicado somente a propriedades de objetos [9].

Conforme o Exemplo 14, sendo a propriedade *subRegionOf* uma propriedade transitiva, o indivíduo *Paraná* está associado ao indivíduo *Brasil*, devido o mesmo estar associado ao indivíduo *Sul*, que por outro lado está associado ao indivíduo *Brasil*.

Exemplo 14:

```
<owl:TransitiveProperty rdf:ID="subRegionOf">
        <rdfs:domain rdf:resource="#Region"/>
        <rdfs:range rdf:resource="#Region"/>
</owl:TransitiveProperty>

<Region rdf:ID="Paraná">
          <subRegionOf rdf:resource="#Sul"/>
</Region>

<Region rdf:ID="Sul">
          <subRegionOf rdf:resource="#Brasil"/>
</Region>
```

Similarmente o operador *owl:SymmetricProperty* determina que se um par (x, y) é uma instância de *P*, então o par (y, x) também deverá ser uma instância de *P*. Este tipo de propriedade também é aplicada somente as propriedades de objetos [9].

Devido a propriedade ser do tipo simétrica no Exemplo 15, o indivíduo *João* está associado ao indivíduo *Mario*, ocorrendo também o processo inverso.

Exemplo 15:

```
<owl:SymmetricProperty rdf:ID="friendOf">
        <rdfs:domain rdf:resource="#Human"/>
        <rdfs:range rdf:resource="#Human"/>
</owl:SymmetricProperty>
```

<Human rdf:ID="João">
<friendOf rdf:resource="#Mario"/>
</Human>

Nesta seção foram apresentados todos os operadores OWL para representações de propriedades de dados e de objetos definidos pela W3C e que são utilizados neste trabalho.

2.1.3 Restrições

Restrições são conceitos definidos na ontologia conforme o domínio que está sendo modelado. Descrevem uma classe anônima de todos os indivíduos que satisfazem a restrição definida nas suas propriedades, sendo um conceito conectado às propriedades. Conforme representado pelas caixas cinza claro na Figura 2.1, é possível a utilização de dois tipos de restrições em OWL: restrições de valores, sendo modeladas com associações *owl:hasValue*, *owl:someValuesFrom* e *owl:allValuesFrom*, e restrições de cardinalidades, sendo modeladas com associações *owl:minCardinality*, *owl:Cardinality* e *owl:maxCardinality* [9, 31].

As restrições de valores estabalecem os limites de associação entre instâncias de uma determinada classe. Assim, a restrição *owl:allValuesFrom* é utilizada para descrever uma classe de todos os indivíduos que estejam associados a uma propriedade específica, devam ser associados somente a indivíduos da classe indicada por *owl:allValuesFrom* [9].

Conforme o Exemplo 16, todas as instâncias que sejam relacionadas à propriedade *hasParent*, devem ter como instâncias associadas somente indivíduos da classe *Human*.

Exemplo 16:
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:allValuesFrom rdf:resource="#Human" />
</owl:Restriction>

A restrição *owl:someValuesFrom* descreve uma classe de que, no mínimo, um dos indivíduos que estejam associados a uma propriedade específica, deva estar associado a indivíduos da classe indicada por *owl:someValuesFrom* [9].

No Exemplo 17, no mínimo uma instância que utiliza a propriedade *hasParent* deverá estar associada a uma instância da classe *Human*.

Exemplo 17:
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:someValuesFrom rdf:resource="#Human" />
</owl:Restriction>

A restrição *owl:hasValue* descreve uma classe de todos os indivíduos que satisfazem uma determinada propriedade, com no mínimo um valor semanticamente verdadeiro [9].

O Exemplo 18 determina uma classe de todas as instâncias que utilizam a propriedade *hasSouthRegion* e possuam o valor *Paraná* como verdadeiro.

Exemplo 18:
<owl:Restriction>
<owl:onProperty rdf:resource="#hasSouthRegion" />
<owl:hasValue rdf:resource="#Paraná" />
</owl:Restriction>

As restrições de cardinalidade permitem que uma classe tenha um número arbitrário de valores com base em uma determinada propriedade. Assim, a restrição *owl:maxCardinality* descreve uma classe de todos os indivíduos que possuem no máximo, "N" valores semanticamente distintos [9].

Conforme o Exemplo 19, a restrição *owl:maxCardinality* descreve uma classe definindo que todos os indivíduos que fazem uso da propriedade *hasParent*, obrigatoriamente tenham que ter no máximo dois pais.

Exemplo 19:
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">2
</owl:maxCardinality>
</owl:Restriction>

A restrição *owl:minCardinality* descreve uma classe de todos os indivíduos que possuem no mínimo, "N" valores semanticamente distintos [9]. Desse modo, ao contrário do Exemplo 19, o Exemplo 20 descreve uma classe definindo que todos os indivíduos que fazem uso da propriedade *hasParent*, obrigatoriamente tenham que ter no mínimo dois pais.

Exemplo 20:
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">2
</owl:minCardinality>
</owl:Restriction>

A restrição de cardinalidade *owl:cardinality*, descreve uma classe de todos os indivíduos que tem exatamente "N" valores semanticamente distintos [9]. Assim, conforme o Exemplo 21, o construtor *owl:cardinality* descreve uma classe de todos os indivíduos que tenham exatamente dois pais.

Exemplo 21:
<owl:Restriction>
<owl:onProperty rdf:resource="#hasParent" />
<owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">2
</owl:cardinality>
</owl:Restriction>

Nesta seção foram apresentados todos os operadores OWL para representações de restrições definidos pela W3C e que são utilizados neste trabalho.

2.2 Defeitos de modelagem em ontologias OWL

Conforme Delamaro, Maldonado e Jino [20], defeitos podem ser um passo, processo ou definição de dados incorretos que são inseridos durante o desenvolvimento de software, estão associados a um determinado programa ou modelo e não dependem de uma execução particular. Defeitos emergem provocando falhas que por sua vez dependem de dados e execuções específicas.

No que se refere à elaboração de ontologias, como não existe apenas uma forma correta de modelar um domínio do conhecimento específico, diversos tipos de defeitos podem ser cometidos, como a definição incorreta de uma restrição, que pode levar a ontologia à uma falha, podendo muitas vezes invalidar a ontologia em questão.

Os defeitos estão relacionados a características próprias das ontologias e geralmente confundem os modeladores [32, 62]. Assim, Gómez-Perez [32] apresenta uma classificação dos principais defeitos que podem ocorrer ao modelar conceitos e ideias dentro de ontologias, e define três grandes classes para esses defeitos: defeitos de inconsistência, defeitos de incompletude e defeitos de redundância. Esses defeitos são detalhados na Tabela 2.1:

Tabela 2.1: Classificação de defeitos em ontologias [32]

Classe	Tipo de defeito	Descrição
Inconsistência	Defeitos Circulatórios	Ocorre quando uma classe é definida como uma subclasse ou superclasse de si mesma em qualquer nível da hierarquia.
	Defeitos de partição	Ocorre quando 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. Por exemplo, se uma classe *Pessoa* é particionada nas subclasses *Homem* e *Mulher*, e define-se uma classe *Paciente* como subclasse tanto de *Homem* quanto de *Mulher*.
Incompletude	Classificação Incompleta	Ocorre quando se omite alguns dos conceitos presentes no domínio do conhecimento durante a classificação dos subconceitos. Por exemplo, ao classificar o conceito *Pessoa*, define-se somente as classes *Jovem* e *Adulto*, omitindo-se outras possibilidades como *Criança* ou *Idoso*.
	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. Por exemplo, ocorre ao definir duas classes *Homem* e *Mulher* que são subclasses da classe *Pessoa*, mas omite-se que ambas as subclasses são disjuntas. Ao omitir-se a disjunção, permite-se que um indivíduo seja instanciado como sendo pertencente a ambas as classes simultaneamente.
Redundância	Redundância gramatical	Ocorre quando existe mais de uma definição explícita de qualquer uma das relações da hierarquia, tal como redundância de relação de subclasse ou de instância.
	Definição formal idêntica	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.

Outra classificação dos principais defeitos que podem ocorrer ao modelar conceitos e ideias dentro de ontologias pode ser encontrada em Rector et al. [62], onde também classificam os principais erros que podem ocorrer ao modelar uma ontologia:

- **Assumir características implícitas nos nomes dos componentes:** as características dos componentes da ontologia (classes, propriedades ou instâncias) não estão disponíveis automaticamente para o classificador. Por exemplo, a falta de disjunção entre duas classes referenciadas como *Homem* e *Mulher*, é possível instanciar objetos para ambas as classes, pois a informação implícita nos seus nomes não é disponível para o classificador como a definição explícita da disjunção.

- **Uso equivocado de qualificadores de restrições:** há comumente uma confusão entre a aplicação de restrições usando os qualificadores *allValuesFrom* (universal) e *someValuesFrom* (existencial). Por exemplo, no contexto deste trabalho, ao definir uma classe como *Consulta*, define-se que esta pode ter pessoas envolvidas que podem ser tanto instâncias da classe definida como *Medico*, quanto da classe definida como *Paciente*, (*someValuesFrom Medico* e *someValuesFrom Paciente*), mas omite-se o fato de que nenhuma outra pessoa pode estar envolvida em uma *Consulta*, o que não impediria que também pudessem participar instâncias da classe definida como *Enfermeiro*. Devido a isso, a restrição *"allValuesFrom (Medico ou Paciente)"* seria ainda necessária para completar a definição da classe.

- **Open World Assumption:** 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 esta 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. Se para uma classe definida como *MedicoClinicoGeral*, que possui a restrição de equivalência que são médicos que não possuem nenhuma relação com instâncias da classe *Especialidade*, o fato de instanciar um *Medico* sem associá-lo a alguma *Especialidade*, não prova que esse indivíduo não a tenha, ou seja, seria necessário dizer explicitamente que o indivíduo não possui qualquer especialidade.

- **Domínio e Intervalo de Propriedades:** 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. Considerando uma classe definida como *Jovem*, sendo esta uma subclasse da classe *Pessoa*, representada com um critério de equivalência de que são todos os indivíduos com a propriedade *hasIdade* com valor menor a 18. Se esta propriedade não tiver um domínio específico a ser aplicado (a classe *Pessoa*, no caso), a propriedade *hasIdade* poderia ser atribuída a qualquer outro tipo de indivíduo, de outras classes, e o classificador poderia indicar como pertencente a classe *Jovem* não apenas instâncias das classes *Medico*, *Paciente* ou *Pessoas* em geral, mas também qualquer tipo de indivíduo instanciado, como das classes *Atendimento* ou *ProcedimentoClinico*.

- **Utilização incorreta dos operadores AND e OR:** A aplicação incorreta dos operadores *AND* e *OR* apresenta resultados incompletos na aplicação de um classificador na ontologia. No caso da definição de duas classes disjuntas *AtendimentoExame* e *AtendimentoVacina*, ao localizar todos os atendimentos do tipo *exame* e todos os atendimentos do tipo *vacina* utilizando o operador *"OR"*, os atendimentos de ambas as classes seriam localizados, ao contrário do operador *"AND"*, que não localizaria nenhum dos atendimentos por não haver instâncias que correspondessem a ambas as classes simultaneamente.

Conforme os defeitos apresentados, Villalón, Suárez-Figueroa e Gómez-Perez [58], 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. Objetivando eliminar estas práticas catalogaram 24 defeitos ocorridos.

Do mesmo modo, Corcho, Roussey e Blazquez [13], 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.

A definição dos problemas de modelagem abordados, que segundo Corcho, Roussey e Blazquez [13] são definidos como antipadrões devido a maioria dos erros serem causados por padrões adotados por especialistas da área, que normalmente resultam em inconsistências, equivalem-se aos problemas apresentados por Villalón, Suárez-Figueroa e Gómez-Perez [58].

1. **Criação de elementos polissêmicos:** refere-se a um elemento cujo nome tem significados diferentes que é incluído na ontologia para representar mais de uma ideia conceitual. Por exemplo, ao definir uma classe *Agente_de_saúde*, que pode representar tanto indivíduos *médicos* como *enfermeiros*.

2. **Criação de classes sinônimas:** ocorre ao criar duas ou mais classes com identificadores sinônimos e definí-las como equivalentes, como ao representar em uma ontologia as classes *Carros*, *Veículos* e *Automóveis*, definidas como equivalentes, sendo os seus indivíduos os mesmos para ambas as classes. Este defeito também é mencionado por Gómez-Perez [32] como *Definição formal idêntica*.

3. **Não definir elementos primitivos:** 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 do tipo *"is"*, para representar a associação entre os componentes. Por exemplo, ao representar as classes *"Car is Automobile"* ao invés de definir a classe *Car* como subclasse de *Automobile*.

4. **Criação de membros desconexos:** ocorre ao definir componentes na ontologia (classes, propriedades ou indivíduos) sem relação alguma com o resto da ontologia. Por exemplo, ao criar o relacionamento *"membrosDoTime"* e não definir a classe que representa as equipes, assim, a relação criada é isolada na ontologia.

5. **Definição incorreta de propriedades inversas:** ocorre ao definir incorretamente uma propriedade como inversa de outra. Por exemplo, ao representar em uma ontologia que *"algo é vendido"* ou *"algo é comprado"*, e definir as propriedades

"vendidoEm" e *"compradoEm"* como inversas, sendo que as mesmas representam conceitos diferentes.

6. **Inclusão de ciclos na hierarquia:** refere-se à inclusão de duas classes na hierarquia da ontologia para representar uma como superclasse ou subclasse da outra. Por exemplo, no caso onde uma classe "A" possui como subclasse uma classe "B" e, ao mesmo tempo essa classe "B" é definida como superclasse de "A". Este defeito também é citado por Gómez-Perez [32] como *Defeito Circulatório*.

7. **Criar uma classe para representar diferentes conceitos:** ocorre ao criar uma classe com um identificador que refere-se para mais de um conceito. Por exemplo, ao definir uma classe *"Produto_ou_ Servico"* para representar indivíduos diferentes.

8. **Anotações incompletas:** 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.

9. **Informações básicas incompletas:** ocorre quando é deixado de incluir informações necessárias na ontologia. Por exemplo, ao definir uma propriedade *"iniciaEm"* para representar o ponto inicial de uma rota, mas deixar de definir uma propriedade *"terminaEm"* para representar o seu fim.

10. **Falta de disjunção:** ocorre devido a falta da definição de disjunção entre classes ou propriedades da ontologia. Por exemplo, para uma classe *Pessoa* que possui duas subclasses *Homem* e *Mulher*, e estas não estão definidas como disjuntas, podendo um indivíduo ser instanciado para ambas ao mesmo tempo. Este defeito também é mencionado por Gómez-Perez [32] como *Omissão do Conhecimento Disjunto*.

11. **Falta de domínio ou intervalo em propriedades:** ocorre devido ao definir propriedades ou indivíduos na ontologia sem determinar seu domínio ou intervalo de dados, ou ambos. Um exemplo é definir uma propriedade como *escritoPor*, na qual o domínio deveria ser a classe *Escritor*, e o intervalo de dados a classe *ObraLiteraria*,

mas um dos dois ou ambos não são específicados. Este defeito também é mencionado por Rector et al. [62] como *Domínio e Intervalo de Propriedades*.

12. **Falta de equivalência entre propriedades:** 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. Por exemplo, duas classes definidas como *CITY* e *city* de duas ontologias diferentes, são definidas como equivalentes, mas as propriedades *hasMember* e *has-Member* não.

13. **Falta de propriedades inversas:** ocorre quando é omitida a definição de inversão entre duas propriedades. Por exemplo, não especificar que ao definir as propriedades *maeDe* e *filhoDe*, estas sejam inversas.

14. **Uso incorreto da restrição allValuesFrom:** este 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. Por exemplo, ao definir que a classe *Hospital* é composta por indivíduos das classes *Medicos* e *Enfermeiros*, impossibilitando de adicionar indivíduos da classe *Pacientes*. Este defeito também é mencionado por Rector et al. [62] como *Uso Equivocado de Restrições*.

15. **Uso incorreto da definição "some not":** ocorre ao confundir a representação de "some not". Por exemplo, ao definir a classe *PizzaVegetariana* com a seguinte definição de equivalência: *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*.

16. **Uso incorreto de classes primitivas e definidas:** geralmente ocorre devido ao desconhecimento por parte do desenvolvedor de as ontologias serem *Open World*

Assumption, de modo que o fato de não definir uma propriedade para um indivíduo da ontologia, não determina que este indivíduo não faça parte desta propriedade. Por exemplo, ao não definir para um indivíduo de uma classe *Medicos* que deva ser associado ao indivíduo *Cardiologista* através da propriedade *hasEspecialidade*, isto não garantirá que este indivíduo não seja um *cardiologista*.

17. **Alto grau de especialização:** 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. Por exemplo, ao definir uma classe *Lugar* e criar como suas subclasses as classes *Barcelona*, *Madrid* e *Sevilla*, ao invés de definí-las como indivíduos.

18. **Intervalo de dados ou domínio limitado:** ocorre ao limitar o domínio ou intervalo de uma propriedade de forma especificada. Por exemplo, ao definir o domínio da propriedade *idiomaOficial* para a classe *Cidade* ao invés da classe *País*.

19. **Trocar os construtores intersecção e união:** ocorre pelo fato do desenvolvedor especificar o limite ou domínio de uma propriedade pela intersecção de várias classes, onde o correto seria definir pela união destas classes, ou o contrário. Por exemplo, ao definir a propriedade *aconteceEm* com seu domínio para a classe *JogosOlimpicos*, e definir como seu intervalo a intersecção das classes *Cidade* e *Nação*, ao invés da união destas classes. Este defeito também é mencionado por Rector et al. [62] como *Utilização incorreta dos operadores AND e OR*.

20. **Trocar rótulos por comentários:** ocorre devido o desenvolvedor trocar o conteúdo de um comentário de um objeto por um rótulo. Por exemplo, especificar um comentário de uma cláusula específica no construtor *Label* e uma palavra que define uma classe, no construtor *Comment*.

21. **Definir uma classe "Diversos":** 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. Por exemplo, ao

definir a classe *Animais* com suas subclasses *Gato*, *Cachorro* e *Diversos*, onde as instâncias da classe *Diversos* não se enquadram tanto em *Gato* quanto em *Cachorro*.

22. **Utilizar diferentes critérios de nomenclatura:** ocorre quando o desenvolvedor não padroniza o modo de nomear os elementos da ontologia. Por exemplo, ao criar algumas classes com os identificadores definidos com todas as letras em maíusculo, outras com todas em mínusculo e algumas somente com a primeira letra em maíusculo.

23. **Utilizar elementos incorretamente:** é ocasionado na ontologia quando um elemento (classe, propriedade ou indivíduo) que foi usado para modelar uma parte desta ontologia deveria ter sido modelado como um elemento diferente. Por exemplo, ao definir a propriedade *isEcologico* entre uma instância da classe *Carros* e as instâncias *Sim* ou *Nao*, deveria ser criado o atributo *isEcologico* cujo intervalo seria *boolean*.

24. **Definir um elemento recursivamente:** ocorre devido um elemento da ontologia ter sido usado em sua própria definição. Por exemplo, o axioma usado para criar a propriedade *temCobertura*, estabelece como limite a seguinte definição de equivalência: *pizzas que têm, pelo menos, um valor para a propriedade temCobertura*.

2.3 Considerações finais

Neste capítulo foram apresentadas as definições e conceitos para a representação de domínios específicos do conhecimento através de ontologias. Foi apresentado um levantamento dos principais defeitos que ocorrem no momento do desenvolvimento destas ontologias, e que devem ser considerados para a análise e avaliação da corretude destes modelos. Esses defeitos são baseados nas estruturas e modos de implementação destas ontologias.

Estes defeitos serão considerados nesta dissertação de mestrado para a definição dos operadores de mutação, que serão utilizados para a aplicação do teste de mutação em ontologias OWL, simulando os principais erros de desenvolvimento, com a finalidade de avaliar a sua corretude.

CAPÍTULO 3

TESTES

Conforme destacam Myers e Sandler [54], a atividade de teste é o processo de executar um programa com a intenção de revelar defeitos. A atividade de teste é um elemento crítico da garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação [59]. Para um programa em teste, são gerados diversos casos de testes que são compostos de dados de entrada mais saída esperada. Para um teste bem sucedido é necessário que um bom conjunto de casos de teste seja utilizado, de forma a aumentar a probabilidade de revelar um defeito ainda não descoberto [59, 53].

Durante a fase de projeto de um processo de desenvolvimento de software, são criados diagramas e modelos que representam diferentes visões do sistema. De forma análoga, o teste sistemático desses modelos também pode revelar defeitos, antes mesmo da fase de desenvolvimento propriamente dita [43].

De acordo com Delamaro, Maldonado e Jino [20], a construção de software depende principalmente da habilidade, da interpretação e da execução das pessoas que o constroem, por isso, erros acabam surgindo, mesmo com a utilização de métodos e ferramentas de engenharia de software.

Assim, o teste de software é um processo da atividade de teste, no qual um programa é executado com casos de teste para determinar se ele atingiu suas especificações e funciona corretamente no ambiente para o qual foi projetado, de acordo com algum critério de teste.

Deste modo, após definido o conjunto de casos de teste T, executa-se o programa P em teste com T e verifica-se qual é o resultado obtido. Se o resultado produzido pela execução de P coincide com o resultado esperado, então nenhum erro foi identificado. Se, para algum caso de teste, o resultado obtido difere do resultado esperado, então um defeito foi revelado [20].

3.1 Teste de software

Um processo de teste de software envolve planejar, projetar, executar casos de teste e avaliar os resultados obtidos. A etapa de elaboração de casos de testes é uma das mais difíceis de realizar, pois dela depende a qualidade dos dados gerados [27].

Conforme Delamaro, Maldonado e Jino [20], um dado de teste para um programa P é um elemento do conjunto de todos os possíveis valores que podem ser utilizados para executar P, ou seja, é um elemento do domínio de entrada D de P, denotado por $D(P)$. Já um caso de teste, é um par formado por um dado de teste mais o resultado esperado para a execução de P com aquele dado de teste. O resultado esperado corresponde ao resultado que P deveria produzir conforme a sua especificação, considerando a entrada fornecida.

Para garantir que o programa não contém defeitos, o ideal seria executar um conjunto de casos de teste com todos os elementos possíveis de $D(P)$, mas devido ao tamanho de $D(P)$, isto pode se tornar impraticável. Deste modo são requeridos critérios de seleção de casos de teste que tenham alta probabilidade de revelar a presença de defeitos caso eles existam [20]. É importante utilizar um subconjunto reduzido de $D(P)$, produzindo assim subdomínios de casos de testes equivalentes [20]. Um critério de seleção de casos de teste deve requerer um elemento de cada subdomínio.

Assim, existem dois modos de selecionar elementos para um subconjunto de teste: o primeiro é o teste aleatório, em que um grande número de casos de teste é selecionado aleatoriamente [20], e o segundo é utilizando critérios de seleção de casos de teste.

Não existe, em princípio, nenhuma restrição sobre o tipo de técnicas de teste que podem ser definidas e aplicadas. Assim, três técnicas são utilizadas para o processo de teste: a funcional, onde os dados de teste são gerados considerando os requisitos funcionais do sistema [20]; a estrutural, onde os dados de teste são gerados considerando a estrutura interna do programa em teste [59]; e a técnica baseada em defeitos, que consiste em analisar os erros que são comumente cometidos durante o processo de desenvolvimento.

Conforme exposto por Rapps e Weyuker [61], um critério de teste estabelece requisitos a serem satisfeitos pelos casos de teste e pode auxiliar na seleção de dados e na avaliação

de sua qualidade. Satisfazer um critério significa atender aos requisitos estabelecidos por ele. Para a definição de um critério de teste específico, Wong, Mathur e Maldonado [71], expõem algumas restrições: custo para a execução, que avalia o número de casos de teste que será necessário criar para satisfazê-lo; eficácia do critério, ou seja, a capacidade de revelar um número maior de defeitos e dificuldade de satisfação, que indica a probabilidade de satisfazer um critério tendo satisfeito outro.

Estudos da literatura têm mostrado que o teste de mutação é o mais eficaz em revelar defeitos, porém o mais custoso [71].

3.2 Teste de mutação em programas

O teste de mutação é uma técnica baseada em defeitos, onde é analisado o código fonte de um programa P a ser testado [46]. Nessa técnica, pequenas alterações sintáticas são introduzidas em P através dos operadores de mutação, que geram novos programas P' chamados mutantes [21]. São utilizados casos de teste T na execução destes mutantes P' para distingui-los do programa original P, de modo que cada mutante P' é executado com o mesmo conjunto de casos de teste T, aplicados 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 dada linguagem, bem como os enganos mais comuns cometidos durante a sua utilização [20].

Além dos tipos de erros 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 [20]. Em geral, os operadores de mutação estão associados a um tipo ou classe de erros que se pretende revelar em P [26]. Alguns exemplos de operadores para a linguagem C [3] são apresentados na Tabela 3.1.

O teste de mutação exige que todos os mutantes sejam mortos [21], um programa mutante P' é dito morto quando um caso de teste conseguir fazer a distinção entre P' e P, gerando saídas diferentes. Se a saída de P for considerada correta, então este estará livre do possível defeito descrito por P'.

Tabela 3.1: Exemplos de operadores de mutação [3]

Operador de Mutação	Definição	Descrição
SSDL	Eliminição de comandos	Projetado para mostrar que cada comando do programa tem um efeito sobre a saída calculada, assim, diversos comandos são retirados do programa P para gerar os mutantes.
ORRN	Troca de operador relacional	Atua sobre cada expressão do programa que contenha algum operador relacional, trocando-o por todos os outros, deste modo, operadores como $>$ são substituídos por $<$ ou $\leq$.
Vsrr	Troca de variáveis escalares	Cada referência a uma variável não estruturada é substituída pela referência a todas as outras variáveis escalares do programa, uma de cada vez.
OPPO	Substitui operadores de incremento e decremento	Cada referência de incremento à uma variável é alterada por uma referência de decremento, ou o processo inverso, assim, se a variável é incrementada antes da sua atribuição, a referência é alterada para depois da sua atribuição.

Os casos de teste que matam os mutantes são classificados como eficientes. Caso, após a execução de todos os casos de testes, ainda existirem mutantes que gerem a mesma saída de P, e se não for possível gerar um caso de teste cuja saída diferencie P de P', os mutantes são considerados equivalentes a P [27], no sentido de que, ou P está correto, ou possui erros pouco prováveis de ocorrerem [21]. Define-se que um conjunto de casos de teste T é adequado para um programa P em relação a P', se para cada programa Q pertencente a P', ou Q é equivalente a P ou Q difere de P em pelo menos um caso de teste [20]. A determinação de equivalência é sempre realizada pelo testador, pois determinar se dois programas computam a mesma função é uma questão indecidível [10]. Essa limitação teórica, não significa que o problema deva ser abandonado por não apresentar solução.

Uma adequação dos casos de teste T 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 [20, 21].

Para um programa **P** e um conjunto de casos de teste **T**, o escore de mutação **S** é obtido da seguinte maneira [20]:

$$\mathrm{S(T)} = \frac{Mm}{(Mg - Me)}, \text{ onde:}$$

- 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 eficaz se torna esse conjunto.

3.3 Teste em ontologias OWL

Métodos de teste que validem os modelos representados em OWL são importantes para garantir adequação e qualidade no desenvolvimento de ontologias, permitindo a representação de domínios de conhecimento com segurança e veracidade.

Da mesma forma que critérios podem ser propostos para selecionar casos de teste, ou então, medir a adequação dos modelos UML em um projeto de software [4], as ontologias, enquanto modelos que representam um domínio de conhecimento, também podem ser executadas e testadas, podendo ser validadas a partir de critérios estabelecidos.

A Tabela 3.2 aponta 6 tipos de testes que podem ser aplicados a ontologias [29].

Tabela 3.2: Tipos de testes aplicados a ontologias [29]

Tipo de Teste	**Descrição**
Instanciação	Especifica se um indivíduo pertence a uma dada classe.
Recuperação	Especifica as instâncias que devem pertencer a uma classe.
Realização	Especifica as classes que os indivíduos serão instanciados.
Satisfação	Especifica se uma inconsistência deve ocorrer depois de adicionar uma nova instancia em uma classe.
Classificação	Determina as classes que um indivíduo deve pertencer.
Consultas escritas em (*SPARQL* e *DL Query*)	Os resultados das consultas são associados com as condições e os critérios projetados.

Tipos de testes baseados em consultas utilizam-se de Questões de Competência (do inglês, *Competency Questions - CQ*), que descrevem os tipos de conhecimento que a suposta ontologia retorna ao responder tais questões. Estas questões devem ser formalizadas em uma linguagem de consulta, para que sistemas testem automaticamente se a ontologia relaciona os resultados com as questões de competência [68]. Se por um lado tais *CQs* são o núcleo da especificação funcional da ontologia, por outro lado, são um conjunto de dados de testes que a ontologia deve executar [29].

Exemplos de questões de competência são, "quais os principais sintomas apresentados por pacientes?" ou "que tipos de reações alérgicas são constatadas em gestantes?". Tais *CQs* devem ser elaboradas em linguagens de consultas para ontologias, como por exemplo *SPARQL* e *DL Query*.

Conforme Vrandecic e Gangemi [68], 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.

Para completar a definição de um teste de uma ontologia, o especialista do domínio deve especificar qual é o resultado esperado. Algumas vezes tais resultados podem ser simplesmente *True* ou *False*, em outros casos podem ser uma classe, lista de classes, uma instância ou uma lista de instâncias [29].

Para os outros tipos de teste da Tabela 3.2 é possível utilizar a lógica baseada em regras, que são construídas na linguagem da ontologia. Por exemplo, objetos em uma ontologia que são ditos diferentes um do outro e a ontologia não consegue informar que eles são a mesma coisa, ou quando duas classes são ditas disjuntas e a ontologia não pode ter declarações que mencionem uma instância como sendo membro de ambas as classes [65].

3.4 Considerações finais

Neste capítulo foram apresentados os conceitos de teste de software, evidenciando a sua importância para a descoberta de erros mais comuns cometidos por programadores du-

rante o desenvolvimento de programas. Neste contexto, o teste de mutação em programas é destacado por ser utilizado nesta dissertação de mestrado para o teste de ontologias OWL, com o objetivo de revelar os principais defeitos cometidos durante o desenvolvimento.

Alguns métodos apresentados para testes em ontologias serão utilizados neste trabalho em conjunto com o teste de mutação. Por exemplo, a utilização de questões de competência como dados de teste a serem aplicados sobre as ontologias.

CAPÍTULO 4

TESTE DE MUTAÇÃO EM ONTOLOGIAS OWL

Neste trabalho é utilizado o teste de mutação para testar ontologias OWL, de modo que operadores de mutação não precisam ser aplicados necessariamente a algum tipo de código implementado. Diversos tipos de operadores voltados para modelos orientados a objetos são propostos por Derezinska [23], dentre eles, os operadores que realizam mutações nas hierarquias entre as classes, também mencionados por Lee, Bai e Chen [46], como mutação de superclasses e subclasses. Neste trabalho estes operadores foram utilizados como base para a definição de novos operadores de mutação para serem aplicados a ontologias OWL, dada a característica de hierarquias deste tipo de modelo de definição. Lee, Bai e Chen [46], propõem alguns operadores de mutação para o teste de *Web Services* baseados em ontologias OWL-S:

- **Classes equivalentes:** alterar as definições de equivalência de uma classe, adicionando, removendo ou substituindo uma dessas definições.
- **Classes disjuntas:** alterar as definições de classes disjuntas, podendo adicionar, remover ou substituir essas definições em uma classe.
- **Classes complexas:** realizar mutação da definição de classes que se utilizam de expressões, realizando alteração na junção dessas expressões (união, intersecção e negação - *AND*, *OR* e *NOT* respectivamente).
- **Classes enumeradas:** referem-se à classes definidas a partir da enumeração de indivíduos que farão parte desta classe, podendo-se adicionar, remover ou substituir instâncias dessas definições.

4.1 Visão geral

Devido à similaridade nos modelos de representação de ontologias com modelos orientados a objetos, tais como classes em um diagrama de classes UML, os operadores de mutação propostos para sistemas orientados a objetos [46, 23], podem ser adaptados a estes modelos de representação do conhecimento.

Deste modo, dada uma ontologia O a ser testada, define-se um conjunto de casos de teste T, cuja qualidade se deseja avaliar, onde T deve ser definido como consultas escritas em uma linguagem de ontologia, como *DL Query* ou *SPARQL*. Assim, T é executado em O e caso apresente resultados incorretos, um defeito é encontrado e o teste é finalizado.

Caso defeitos não sejam encontrados no processo anterior, O ainda pode conter defeitos que não foram revelados pelo conjunto de casos de teste T. Assim, O sofre pequenas alterações em sua estrutura, dando origem às ontologias mutantes O'.

As alterações realizadas em O são geradas por operadores de mutação pré-definidos. Cada operador realiza uma mudança sintática simples com base nos erros típicos cometidos pelos programadores, ou com o objetivo de forçar determinados objetivos de teste.

Para aplicar o teste de mutação são estabelecidos os seguintes processos:

- geração das ontologias mutantes;
- geração dos dados de teste;
- teste da ontologia original;
- teste das ontologias mutantes;
- análise dos resultados.

As definições dos operadores de mutação propostos, assim como a descrição e um exemplo de aplicação são descritos na seção 4.2, do mesmo modo, a execução dos passos para aplicar o teste de mutação em ontologias OWL, é definido nas seções 4.3.1 à 4.3.5.

4.2 Definição dos operadores de mutação

Conforme mencionado no Capítulo 3, não existe uma maneira direta de definir os operadores de mutação, deste modo, tais operadores foram determinados com base nas regras da linguagem OWL e nos enganos mais comuns cometidos durante o desenvolvimento de ontologias, conforme apresentado no Capítulo 2.

Dentro desta perspectiva, propôs-se um conjunto de 25 operadores para classes, restrições e propriedades das ontologias OWL, realizando modificações na definição de restrições e hierarquias de classes e nas restrições de domínio e limite das propriedades.

A definição e objetivo dos operadores de mutação propostos neste trabalho é apresentada a seguir, assim como a descrição e um exemplo de aplicação de cada operador de mutação.

4.2.1 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:** Através do operador *ClassUpCascade*, uma classe C, subclasse de SP_2, sobe um nível na hierarquia e passa a ficar associada imediatamente a superclasse de sua superclasse original *(SP_1)*; suas subclasses SB_n acompanham esta mudança, ou seja, permanecem associadas à classe C.

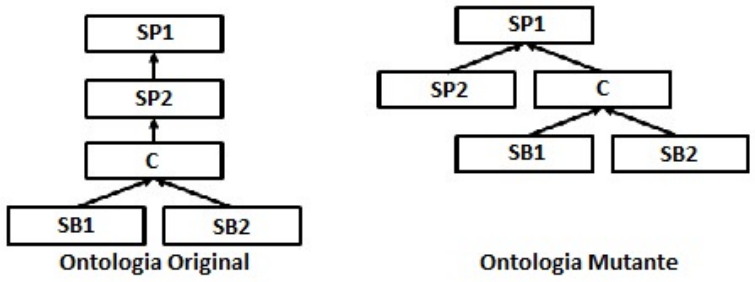

Figura 4.1: Modelo de mutação com o operador ClassUpCascade

- **Exemplo:** *Profissional* é subclasse de *Pessoa* que é subclasse de *Thing*, *Profissional* tem ainda duas subclasses: *Enfermeiro* e *Medico*. *Profissional* sobe na hierarquia e passa a ser subclasse de *Thing*, e suas subclasses *Medico* e *Enfermeiro* continuam sendo subclasses de *Profissional*.

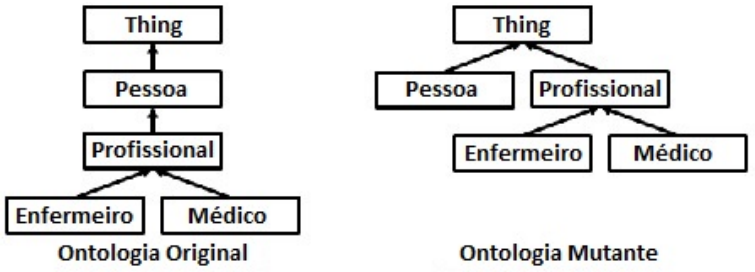

Figura 4.2: Exemplo de mutação com o operador ClassUpCascade

Os defeitos que possivelmente revelam-se com este operador são vistos na Tabela 4.1.

Tabela 4.1: Possíveis defeitos revelados pelo operador *ClassUpCascade*

Defeito	**Motivo**
Defeito circulatório	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	Ao realizar a mutação na hierarquia das classes, é possível que a classe modificada torne-se subclasse de outras duas classes, podendo invalidar a ontologia.
Definição de elementos recursivos	Caso uma classe torne-se superclasse ou subclasse dela mesma, este defeito pode ser revelado devido a ocorrência da recursão apresentada no componente.

4.2.2 Operador de mutação ClassUp - CUP

- **Objetivo:** Alterar a estrutura hierárquica de uma classe alterando também 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 entre C e SP_1 (C passa a ser subclasse de SP_1); 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 alterada para uma relação *subClassOf* entre SB_n e SP_2 (SB_n deixa de ser subclasse de C e passa a ser subclasse de SP_2).

- **Descrição:** O operador *ClassUp* é uma extensão do operador *ClassUpCascade*, pois além de alterar o nível hierárquico de uma classe C também altera a hierarquia das subclasses associadas a C, fazendo com que deixem de ser subclasses de C e passem a ser subclasses da classe que era originalmente a superclasse de C (SP_2).

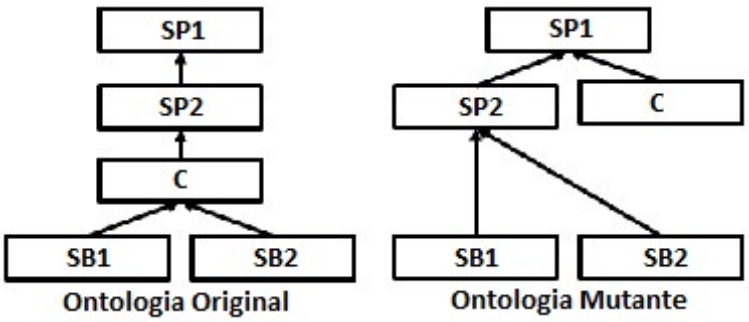

Figura 4.3: Modelo de mutação com o operador ClassUp

- **Exemplo:** *Profissional* é uma subclasse de *Pessoa* que é subclasse de *Thing*, *Profissional* tem ainda duas subclasses: *Enfermeiro* e *Medico*. *Profissional* sobe na hierarquia e passa a ser subclasse de *Thing*, mas suas subclasses *Medico* e *Enfermeiro* continuam sendo subclasses de *Pessoa*.

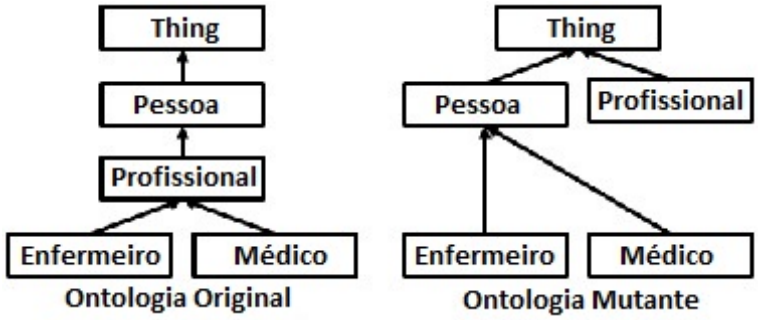

Figura 4.4: Exemplo de mutação com o operador ClassUp

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.1 da Seção 4.2.1, devido as mutações serem similares às realizadas pelo operador de mutação *ClassUpCascade*.

4.2.3 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 *OWL*, dada uma classe *C*, tal que $C \in OWLClass$, para toda classe 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.
- **Descrição:** O operador *ClassDisjointDef* cria uma definição de disjunção entre classes, para todas as subclasses SB_n de uma classe *C*.
- **Exemplo:** As classes *Enfermeiro* e *Medico* são subclasses da classe *Profissional* e não estão definidas como classes disjuntas; essas classes passam a ser disjuntas invalidando qualquer pessoa instanciada para as classes *Medico* e *Enfermeiro* simultaneamente.

Os defeitos de modelagem de ontologias que possivelmente podem ser revelados com este operador de mutação são apresentados na Tabela 4.2.

Tabela 4.2: Possíveis defeitos revelados pelo operador *ClassDisjointDef*

Defeito	Motivo
Classificação incompleta	Ao definir uma restrição de disjunção entre classes da ontologia, pode-se inferir que houve a omissão deste conceito na definição das classes da ontologia original.
Falta de disjunção	Conforme definido para o defeito acima, ao definir uma restrição de disjunção entre classes da ontologia, infere-se que houve a omissão deste conceito na definição das classes da ontologia original.
Informações básicas incompletas	Ao definir uma restrição de disjunção entre classes da ontologia, infere-se que esta definição é uma característica básica da classe sendo mutada.

4.2.4 Operador de mutação ClassDisjointDefOne - CDDO

- **Objetivo:** Adicionar uma definição de disjunção entre duas classes.
- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, para toda classe SB_n, tal que $SB_n \in OWLClass$, 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 duas classes SB_n.
- **Descrição:** O operador *ClassDisjointDefOne* cria uma definição de disjunção entre classes, para duas subclasses SB_n de uma classe C.
- **Exemplo:** *Enfermeiro*, *Medico* e *Recepcionista* são subclasses de *Profissional* e não estão definidas como classes disjuntas; *Enfermeiro* e *Medico* passam a ser disjuntas invalidando qualquer pessoa instanciada como *Medico* e *Enfermeiro* simultaneamente, mas validando a instanciação de qualquer pessoa como *Enfermeiro* e *Recepcionista*, ou *Medico* e *Recepcionista* simultaneamente.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.2 da Seção 4.2.3, devido as mutações serem similares as realizadas pelo operador de mutação *ClassDisjointDef*.

4.2.5 Operador de mutação ClassDisjointUndefOne - CDUO

- **Objetivo:** Remover uma definição de disjunção entre duas classes.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, para toda classe SB_n, tal que $SB_n \in OWLClass$, existe relação *disjointWith* entre as classes SB_n, e o conjunto de classes SB_n é composto por 3 classes ao menos, então a relação *disjointWith* entre as classes SB_n é alterada, removendo-se uma das classes dessa relação.
- **Descrição:** O operador *ClassDisjointUndefOne* altera uma definição de disjunção entre um conjunto de 3 ou mais classes, removendo uma das classes da definição de disjunção.
- **Exemplo:** As classes que representam os diferentes tipos de atendimento (*Consulta*, *Exame* e *Vacina*) estão definidas como disjuntas entre si. Uma das classes é eliminada da disjunção, *Vacina* por exemplo, permanecendo apenas *Consulta* e *Exame* como disjuntos. Além de permitir que um atendimento do tipo *Consulta* ou *Exame* também possam ser classificados como do tipo *Vacina*, a ontologia não reconhece mais os atendimentos do tipo *Vacina* como (*not Consulta*) ou (*not Exame*), tampouco *Consulta* ou *Exame* como (*not Vacina*).

Este operador de mutação, possivelmente pode revelar o defeito de modelagem de ontologias "Assumir características implícitas nos nomes dos componentes", pois ao remover a definição de disjunção de uma classe, desde que esta classe não possua qualquer tipo de restrição de equivalência, seu nome passa a ser o seu único identificador.

4.2.6 Operador de mutação ClassDisjointUndefAll - CDUA

- **Objetivo:** Remover uma definição de disjunção entre um grupo de classes.
- **Definição:** Considerando o conjunto de classes *OWLClass* definido no metamodelo *OWL*, para toda classe SB_n, tal que $SB_n \in OWLClass$, e existe relação *disjointWith*

entre as classes SB_n, então a relação *disjointWith* entre as classes SB_n é removida da definição da ontologia.

- **Descrição:** O operador *ClassDisjointUndefAll* elimina completamente do modelo uma definição de disjunção entre classes.

- **Exemplo:** *Homem* e *Mulher* são classes definidas como disjuntas entre si; ao remover a disjunção não somente um indivíduo da classe *Homem* passa a poder também ser instância da classe *Mulher*, como a ontologia não reconhece mais as expressões (*not Homem*) e (*not Mulher*).

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados pelo operador de mutação *ClassDisjointUndefOne* da Seção 4.2.5, devido as mutações realizadas serem similares à ambos.

4.2.7 Operador de mutação ClassEquivalentUndef - CEU

- **Objetivo:** Remover uma definição de equivalência entre classes.

- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, e existe relação do tipo *equivalentClass* entre C e outras classes do modelo, então essa relação *equivalentClass* pode ser removida da definição da ontologia.

- **Descrição:** O operador *ClassEquivalentUndef* remove do modelo uma definição de equivalência que define uma classe C.

- **Exemplo:** A classe *Gestante* tem uma definição de equivalência *"Pessoa and (isGestante value true)"*. Ao remover essa definição as *Pessoas* que possuem a propriedade *isGestante* com valor *TRUE* não serão mais classificadas como *Gestante*.

Os defeitos de modelagem de ontologias que possivelmente podem ser revelados com este operador de mutação são apresentados na Tabela 4.3.

Tabela 4.3: Possíveis defeitos revelados pelo operador *ClassEquivalentUndef*

Defeito	Motivo
Assumir características implícitas nos nomes dos componentes	Ao remover a definição de equivalência de uma classe, pode-se identificar este defeito, de modo que apenas o seu nome permanece como seu identificador.
Uso incorreto de classes primitivas e definidas	Como as ontologias são *Open World Assumption*, ao remover a definição de equivalência de uma classe, este defeito pode ser revelado, caso esta classe continue sendo instanciada similarmente ao modelo antes da mutação.
Redundância gramatical	Ao remover uma restrição de equivalência de uma classe, este defeito pode ser encontrado caso a mutação torne a classe mutada similar a outra classe definida na ontologia, ou se for perceptível que esta definição torna a classe redundante.
Criação de elementos polissêmicos	Com a remoção de uma restrição de equivalência, pode ocorrer que a classe mutada não consegue instanciar objetos que são obrigatórios a ela, ou pelo contrário, passa a instanciar objetos que não deveriam fazer parte dela.
Criar uma classe para representar diferentes conceitos	Ao remover uma definição de equivalência, é possível que diferentes objetos sejam instanciados para a mesma classe, passando a representar diferentes conceitos.

4.2.8 Operador de mutação ClassEquivalentDef - CED

- **Objetivo:** Adicionar uma definição de equivalência entre classes.

- **Definição:** Dada duas classes C e D, tal que $C \in OWLClass$ e $D \in OWLClass$, e as duas classes estão no mesmo nível da hierarquia da ontologia, e não existe relação *equivalentClass* entre C e D, então uma relação *equivalentClass* pode ser definida entre as classes C e D.

- **Descrição:** O operador *ClassEquivalentDef* adiciona no modelo uma definição de equivalência entre uma classe C e uma classe D, desde que C e D já não possuam. Essa alteração, além de alterar os indivíduos que serão classificados como pertencentes as classes C e D, também pode alterar a estrutura hierárquica de classes do modelo, já que as definições de equivalência implicam em definições automáticas de relação do tipo *complementOf*.

- **Exemplo:** As classes *Gestante* e *Lactante* estão no mesmo nível hierarquico da

ontologia. Ao definir que as duas classes são equivalentes todas as pessoas classificadas como pertencentes a classe *Gestante*, passarão também a ser classificadas como pertencentes a classe *Lactante*, o processo inverso também ocorrerá.

Os defeitos de modelagem de ontologias que possivelmente podem ser revelados com este operador de mutação são apresentados na Tabela 4.4.

Tabela 4.4: Possíveis defeitos revelados pelo operador *ClassEquivalentDef*

Defeito	**Motivo**
Falta de equivalência entre propriedades	Ao adicionar a definição de equivalência entre duas classes, afeta diretamente a instanciação dos objetos, revelando-se o defeito caso o resultado obtido entre as ontologias original e mutante sejam iguais.
Informações básicas incompletas	Ao adicionar a definição de equivalência entre duas classes, infere-se que a ontologia esta incompleta, revelando-se o defeito caso o resultado obtido entre as ontologias original e mutante sejam iguais.

4.2.9 Operador de mutação ClassEquivalentCopyOne - CECO

- **Objetivo:** Duplicar a definição de equivalência entre classes.
- **Definição:** Dada duas classes C e D, tal que $C \in OWLClass$ e $D \in OWLClass$, e as duas classes encontram-se no mesmo nível da hierarquia da ontologia, e existe relação do tipo *equivalentClass* entre C e outras classes do modelo, então essa relação *equivalentClass* é duplicada para a classe D.
- **Descrição:** O operador *ClassEquivalentCopyOne* duplica no modelo uma definição de equivalência que define uma classe C. Essa alteração, além de alterar os indivíduos que serão classificados como pertencentes a classe D, também pode alterar a estrutura hierárquica de classes do modelo, já que as definições de equivalência implicam em definições automáticas de relação do tipo *complementOf*.
- **Exemplo:** A classe *Gestante* tem uma definição de equivalência *"Pessoa and (isGestante value true)"*. Ao duplicar essa definição para a classe *Lactante*, as *Pessoas*

que possuem a propriedade *isGestante* com valor *TRUE*, serão classificadas como pertencentes também a classe *Lactante*.

Os defeitos que possivelmente revelam-se com este operador são vistos na Tabela 4.5.

Tabela 4.5: Possíveis defeitos revelados pelo operador *ClassEquivalentCopyOne*

Defeito	**Motivo**
Redundância gramatical	Ao duplicar uma definição de equivalência de uma classe, pode-se identificar este defeito caso a classe que receba a nova definição não possua nenhuma outra restrição de equivalência. Sendo assim, os nomes serão a única diferença entre ambas.
Definir um elemento recursivamente	Ao duplicar uma definição de equivalência de uma classe, pode-se identificar este defeito desde que a classe que receba a nova definição esteja sendo usada na definição da equivalência duplicada.

4.2.10 Operador de mutação ClassEquivalentUndefAnd - CEUA

- **Objetivo:** Remover o operador "AND" em uma definição de equivalência.
- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, e existe relação do tipo *equivalentClass* entre C e outras classes do modelo, relação esta descrita de forma lógica na forma (A *and* B), sendo A e B classes do conjunto *OWLClass* ou outras expressões lógicas, então essa relação *equivalentClass* pode ser alterada removendo-se um dos termos ligados pelo operador *"and"*. A relação *equivalentClass* passa então a ser descrita na forma lógica apenas pelas expressões (A) ou (B).
- **Descrição:** Quando uma classe é definida como equivalente por uma expressão lógica na forma (A *and* B), o operador *ClassEquivalentUndefAnd* altera essa expressão lógica que define a equivalência da classe, mantendo a expressão somente como (A) ou (B) apenas.
- **Exemplo:** A classe *Gestante* tem uma definição de equivalência *"Pessoa and (isGestante value true)"*. Essa definição poderia ser alterada para duas possíveis situações:

- *"Pessoa"*: qualquer *pessoa* passaria a ser classificada como pertencente a classe *Gestante*, inclusive os indivíduos pertencentes a classe *Homem*, o que poderia incorrer em inconsistência em relação à disjunção das classes *Homem* e *Mulher*.
- *"(isGestante value true)"*: todos os indivíduos instanciados com a propriedade *isGestante* com valor *TRUE* seriam classificados como pertencentes a classe *Gestante*, independente de serem da classe *Pessoa* ou não.

Os defeitos de modelagem de ontologias que possivelmente revelam-se com este operador de mutação são apresentados na Tabela 4.6.

Tabela 4.6: Possíveis defeitos revelados pelo operador *ClassEquivalentUndefAnd*

Defeito	**Motivo**
Uso incorreto de classes primitivas e definidas	Como as ontologias são *Open World Assumption*, ao remover o operador *AND* da definição de uma restrição de equivalência, este defeito pode ser revelado caso esta classe continue sendo instanciada similarmente ao modelo antes da mutação.
Utilização incorreta dos operadores AND e OR	Ao remover o operador *AND* da definição de equivalência de uma classe, este defeito pode ser revelado caso esta classe continue apresentando resultados similares ao modelo antes da mutação.
Redundância gramatical	Ao remover o operador *AND* de uma restrição de equivalência, este defeito pode ser revelado caso a nova definição torne-se similar a outra já definida na ontologia.
Criação de elementos polissêmicos	Com a remoção do operador *AND* de uma restrição de equivalência, pode ocorrer que a classe mutada não consegue instanciar objetos que são obrigatórios a ela, ou pelo contrário, passa a instanciar objetos que não deveriam fazer parte dela, revelando deste modo este defeito.
Criação de classes sinônimas	Ao remover o operador *AND* de uma restrição de equivalência, este defeito pode ser revelado caso a classe torne-se similar a outra já definida na ontologia, de modo que ambas sejam equivalentes.
Criar uma classe para representar diferentes conceitos	Ao remover o operador *AND* de uma definição de equivalência é possível revelar este defeito, pois diferentes tipos de objetos podem ser instanciados para essa classe, passando assim a representar diferentes conceitos.

4.2.11 Operador de mutação ClassEquivalentUndefOr - CEUO

- **Objetivo:** Remover o operador "OR" em uma definição de equivalência.
- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, e existe relação do tipo *equivalentClass* entre C e outras classes do modelo, relação esta descrita de forma lógica na forma $(A\ or\ B)$, sendo A e B classes do conjunto *OWLClass* ou outras expressões lógicas, então essa relação *equivalentClass* pode ser alterada removendo-se um dos termos ligados pelo operador *"or"*. A relação *equivalentClass* passa então a ser descrita na forma lógica apenas pelas expressões (A) ou (B).
- **Descrição:** Quando uma classe é definida como equivalente por uma expressão lógica na forma $(A\ or\ B)$, o operador *ClassEquivalentUndefOr* altera essa expressão lógica que define a equivalência da classe, mantendo a expressão somente como (A) ou (B) apenas.
- **Exemplo:** A classe *Adulto* possui definição de equivalência *"Adulto and ((hasCaracteristica value Menina) or (hasCaracteristica value Menino))"*. Essa definição seria alterada para *"Adulto and (hasCaracteristica value Menina)"* ou *"Adulto and (hasCaracteristica value Menino)"* o que iria influenciar diretamente a classificação de indivíduos nesta classe.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.6 da Seção 4.2.10, devido as mutações serem similares as realizadas pelo operador de mutação *ClassEquivalentUndefAnd*.

4.2.12 Operador de mutação PropertyDomainDown - PDD

- **Objetivo:** Alterar a definição de domínio nas propriedades 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 uma definição de 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 excessivamente genérica para a propriedade em questão.

- **Exemplo:** A propriedade *hasPessoaEnvolvida* possui seu domínio definido com a classe *Atendimento*. O domínio poderia ser alterado para qualquer uma de suas subclasses, como *Consulta*, *Exame* ou *Vacina*, o que faria com que apenas um destes subtipos pudesse ter a propriedade definida em seu escopo.

Os defeitos de modelagem de ontologias possivelmente revelados por este operador de mutação são: "Domínio e intervalo de propriedades", devido a alteração do domínio de uma classe poder causar mudanças estruturais e de instanciação de objetos e, "Intervalo de dados ou domínio limitado", pois ao alterar o domínio de uma classe é possível a obtenção de novos resultados na ontologia.

4.2.13 Operador de mutação PropertyDomainUp - PDUP

- **Objetivo:** Alterar a definição de domínio nas propriedades para uma classe um nível 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 uma definição de domínio de uma propriedade por uma superclasse da classe originalmente definida como sendo este domínio, com o objetivo de testar se essa definição não foi feita de forma excessivamente limitada para a propriedade em questão.

- **Exemplo:** A propriedade *hasPessoaEnvolvida* possui seu domínio definido com a classe *Consulta*. O domínio poderia ser alterado para a sua superclasse *Atendimento*, o que faria com que as outras subclasses, como *Exame* ou *Vacina*, passassem a fazer parte desta propriedade.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Seção 4.2.12, devido as mutações serem similares as realizadas pelo operador de mutação *PropertyDomainDown*.

4.2.14 Operador de mutação PropertyRangeDown - PRD

- **Objetivo:** Alterar a definição de limite nas propriedades para uma classe um nível abaixo na hierarquia.

- **Definição:** 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 limite de valores possíveis da propriedade P passa a ser restrito a C_2).

- **Descrição:** Este operador troca uma definição de limite de uma propriedade por uma subclasse da classe originalmente definida como sendo este limite, com o objetivo de testar se essa definição não foi feita de forma excessivamente genérica para a propriedade em questão.

- **Exemplo:** A propriedade *hasPessoaEnvolvida* possui seu limite definido com a classe *Pessoa*. O limite poderia ser alterado para qualquer uma de suas subclasses,

como *Homem*, *Mulher*, *Gestante*, *Adulto*, *Profissional*, restringindo os indivíduos que poderiam ser utilizados na associação desta propriedade com a classe de domínio.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Seção 4.2.12, devido as mutações serem similares as realizadas pelo operador de mutação *PropertyDomainDown*.

4.2.15 Operador de mutação PropertyRangeUp - PRUP

- **Objetivo:** Alterar a definição de limite nas propriedades para uma classe um nível acima na hierarquia.

- **Definição:** 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 (o limite de valores possíveis da propriedade P passa a ser restrito a C_2).

- **Descrição:** Este operador troca uma definição de limite de uma propriedade pela superclasse da classe originalmente definida como sendo este limite, com o objetivo de testar se essa definição não foi feita de forma excessivamente limitada para a propriedade em questão.

- **Exemplo:** A propriedade *hasPessoaEnvolvida* possui seu limite definido com a classe *Homem*. O limite poderia ser alterado para a sua superclasse *Pessoa*, deste modo todos os indivíduos das outras subclasses, como *Mulher*, *Gestante*, *Adulto*, *Profissional*, poderiam ser utilizados na associação desta propriedade com a classe de domínio.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Seção 4.2.12, devido as mutações serem similares as realizadas pelo operador de mutação *PropertyDomainDown*.

4.2.16 Operador de mutação PropertyDomainUndef - PDU

- **Objetivo:** Remover a definição de domínio nas propriedades.
- **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:** Este operador remove a definição de domínio de uma propriedade, com o objetivo de testar se essa definição não foi feita de forma genérica.
- **Exemplo:** A propriedade *hasPessoaEnvolvida* possui seu domínio definido com a classe *Atendimento*. O domínio é removido, possibilitando que as outras classes do modelo possam fazer parte desta propriedade.

Os defeitos de modelagem de ontologias que possivelmente podem ser revelados com este operador de mutação são apresentados na Tabela 4.7.

Tabela 4.7: Possíveis defeitos revelados pelo operador *PropertyDomainUndef*

Defeito	Motivo
Assumir características implícitas nos nomes dos componentes	Ao remover a definição de domínio de uma propriedade, este defeito pode ser revelado devido este componente passar a não ter mais relação direta com nenhum outro componente da ontologia.
Falta de domínio ou intervalo em propriedades	Conforme descrito no defeito acima, ao remover a definição de domínio de uma propriedade, este defeito também pode ser revelado devido esta propriedade passar a não ter mais relação direta com nenhum outro componente da ontologia.
Criação de membros desconexos	Como definido para os defeitos apresentados acima, ao remover a definição de domínio de uma propriedade, esta propriedade passa a não ter mais relação direta com nenhum outro componente da ontologia.
Intervalo de dados ou domínio limitado	Removendo-se o domínio de uma propriedade, este defeito pode ser revelado caso o domínio tenha sido especificado de forma extremamente genérica.

4.2.17 Operador de mutação PropertyRangeUndef - PRU

- **Objetivo:** Remover a definição de limite nas propriedades.
- **Definição:** 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, então a relação *allValuesFrom* entre P e C é removida (o limite de valores possíveis da propriedade P passa a ser qualquer classe definida).
- **Descrição:** Este operador remove uma definição de limite de uma propriedade, com o objetivo de testar se essa definição não foi feita de forma excessivamente genérica para a propriedade em questão.
- **Exemplo:** A propriedade *hasPessoaEnvolvida* possui seu limite definido com a classe *Pessoa*. O limite é removido, deixando de restringir os indivíduos que poderiam ser utilizados na associação desta propriedade com a classe de domínio.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.7 da Seção 4.2.16, devido as mutações serem similares as realizadas pelo operador de mutação *PropertyDomainUndef*.

4.2.18 Operador de mutação PropertyInverseDef - PID

- **Objetivo:** Definir uma propriedade como inversa de outra.
- **Definição:** Dada duas propriedades P e F, tal que $P \in OWLObjectProperty$ e $F \in OWLObjectProperty$, e as duas propriedades estão no mesmo nível da hierarquia da ontologia, então P passa a ser definida como *inverseOf* de F.
- **Descrição:** Este operador adiciona a definição de que uma propriedade é inversa de outra, com o objetivo de testar a similaridade entre as propriedades.
- **Exemplo:** A propriedade *maeDe* é definida como inversa da propriedade *filhoDe*, o que faz com que para o indivíduo A que está associado pela propriedade *filhoDe*

ao indivíduo B, automaticamente B passa a estar associado ao indivíduo A pela propriedade inversa *maeDe*.

Os defeitos que possivelmente revelam-se com este operador sao vistos na Tabela 4.8.

Tabela 4.8: Possíveis defeitos revelados pelo operador *PropertyInverseDef*

Defeito	Motivo
Definição incorreta de propriedades inversas	Ao definir uma propriedade como inversa de outra, este defeito pode ser revelado caso as duas propriedades apresentem conceitos diferentes uma da outra.
Falta de propriedades inversas	Ao contrário do defeito descrito acima, ao definir uma propriedade como inversa de outra, este defeito pode ser revelado caso as duas propriedades apresentem conceitos similares.
Informações básicas incompletas	Conforme descrito no defeito acima, ao definir uma propriedade como inversa de outra, este defeito também pode ser revelado caso as duas propriedades apresentem conceitos similares.

4.2.19 Operador de mutação PropertyInverseUndef - PIUD

- **Objetivo:** Remover a definição de inversão de uma propriedade.
- **Definição:** Dada duas propriedades P e F, tal que P *e* $F \in$ *OWLObjectProperty*, existe relação do tipo *inverseOf* entre P e F, e as duas propriedades estão no mesmo nível da hierarquia da ontologia, então a relação *inverseOf* é removida entre P e F.
- **Descrição:** Este operador remove a definição de que uma propriedade é inversa de outra, com o objetivo de testar a similaridade entre as propriedades.
- **Exemplo:** A propriedade *maeDe* é definida como inversa da propriedade *filhoDe*, ao remover a definição de que as propriedades são inversas, se o indivíduo A está associado pela propriedade *filhoDe* ao indivíduo B, automaticamente B deixa de estar associado à A pela propriedade *maeDe*.

Os possíveis defeitos revelados por este operador são os mesmos apresentados na Tabela 4.8 da Seção 4.2.18, devido as mutações serem similares entre os operadores.

4.2.20 Operador de mutação AxiomChangeOrToAnd - ACOTA

- **Objetivo:** Substituir o operador "OR" pelo operador "AND" em um axioma.
- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, existe relação do tipo *equivalentClass* entre C e outras classes do modelo, relação esta descrita na forma lógica (A *or* B), sendo A e B classes do conjunto $OWLClass$ ou outras expressões lógicas, então essa relação *equivalentClass* é alterada para (A *and* B). A relação *equivalentClass* passa a ser descrita na forma lógica apenas pelo operador (*and*).
- **Descrição:** Quando uma classe é definida como equivalente por uma expressão lógica (A *or* B), o operador *AxiomChangeOrToAnd* altera essa expressão lógica que define a equivalência da classe, substituindo o operador *or* pelo operador *and*.
- **Exemplo:** A classe *Adulto* possui definição de equivalência *"Adulto and ((hasCaracteristica value Menina) or (hasCaracteristica value Menino))"*. Essa definição seria alterada para *"Adulto and ((hasCaracteristica value Menina) and (hasCaracteristica value Menino))"* o que iria influenciar diretamente a classificação de indivíduos nesta classe.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.6 da Seção 4.2.10, devido as mutações serem similares as realizadas pelo operador de mutação *ClassEquivalentUndefAnd*, diferenciando apenas que ao invés de remover o operador *AND* de um axioma, este operador de mutação substitui o operador *OR* pelo operador *AND* em uma definição.

4.2.21 Operador de mutação AxiomChangeAndToOr - ACATO

- **Objetivo:** Substituir o operador "AND" pelo operador "OR" em um axioma.
- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, existe relação do tipo *equivalentClass* entre C e outras classes do modelo, relação esta descrita na forma lógica (A *and* B), sendo A e B classes do conjunto $OWLClass$ ou outras expressões

lógicas, então essa relação *equivalentClass* é alterada para (*A or B*). A relação *equivalentClass* passa a ser descrita na forma lógica apenas pelo operador (*or*).

- **Descrição:** Quando uma classe é definida como equivalente por uma expressão lógica (*A and B*), o operador *AxiomChangeAndToOr* altera essa expressão lógica que define a equivalência da classe, substituindo o operador *and* pelo operador *or*.

- **Exemplo:** A classe *Jovem* possui a definição de equivalência *"Jovem and ((hasCaracteristica value Menina) or (hasCaracteristica value Menino))"*. Essa definição seria alterada para *"Jovem or ((hasCaracteristica value Menina) or (hasCaracteristica value Menino))"* o que iria influenciar diretamente a classificação de indivíduos nesta classe.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.6 da Seção 4.2.10, devido as mutações serem similares as realizadas pelo operador de mutação *ClassEquivalentUndefAnd*, diferenciando apenas que ao invés de remover o operador *AND* de um axioma, este operador de mutação substitui o operador *AND* pelo operador *OR* em uma definição.

4.2.22 Operador de mutação AxiomChangeSomeToAll - ACSTA

- **Objetivo:** Substituir o operador *someValuesFrom* pelo operador *allValuesFrom* em um axioma.

- **Definição:** Dada uma classe *C*, tal que $C \in OWLClass$, existe relação do tipo *equivalentClass* entre *C* e outras classes do modelo, relação esta descrita na forma lógica (*A someValuesFrom B*), sendo *A* e *B* classes do conjunto *OWLClass* ou outras expressões lógicas, então essa relação *equivalentClass* é alterada para (*A allValuesFrom B*). A relação *equivalentClass* passa a ser descrita na forma lógica apenas pelo operador (*allValuesFrom*).

- **Descrição:** Quando uma classe é definida como equivalente por uma expressão lógica na forma (*A someValuesFrom B*), o operador *AxiomChangeSomeToAll* altera

essa expressão lógica que define a equivalência da classe, substituindo o operador *someValuesFrom* pelo operador *allValuesFrom*.

- **Exemplo:** A classe *Atendimento* possui definição de equivalência *"Atendimento and (hasPessoaEnvolvida someValuesFrom Profissional)"*. Essa definição seria alterada para *"Atendimento and (hasPessoaEnvolvida allValuesFrom Profissional)"* o que iria influenciar diretamente a classificação de indivíduos nesta classe.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.6 da Seção 4.2.10, exceto para o defeito de "Utilização incorreta dos operadores *AND* e *OR*". As mutações aplicadas são similares as realizadas pelo operador de mutação *ClassEquivalentUndefAnd*, diferenciando apenas que ao invés de remover o operador *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.

4.2.23 Operador de mutação AxiomChangeAllToSome - ACATS

- **Objetivo:** Substituir o operador *allValuesFrom* pelo operador *someValuesFrom* em um axioma.

- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, existe relação do tipo *equivalentClass* entre C e outras classes do modelo, relação esta descrita na forma lógica (A *allValuesFrom* B), sendo A e B classes do conjunto *OWLClass* ou outras expressões lógicas, então essa relação *equivalentClass* é alterada para (A *someValuesFrom* B). A relação *equivalentClass* passa a ser descrita na forma lógica apenas pelo operador (*someValuesFrom*).

- **Descrição:** Quando uma classe é definida como equivalente por uma expressão lógica na forma (A *allValuesFrom* B), o operador *AxiomChangeAllToSome* altera

essa expressão lógica que define a equivalência da classe, substituindo o operador *allValuesFrom* pelo operador *someValuesFrom*.

- **Exemplo:** A classe *Atendimento* possui definição de equivalência *"Atendimento and ((hasPessoaEnvolvida allValuesFrom Profissional) and (hasPessoaEnvolvida allValuesFrom Paciente))"*. Essa definição seria alterada para *"Atendimento and ((hasPessoaEnvolvida someValuesFrom Profissional) and (hasPessoaEnvolvida someValuesFrom Paciente))"* o que iria influenciar diretamente a classificação de indivíduos nesta classe.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Tabela 4.6 da Seção 4.2.10, exceto para o defeito de "Utilização incorreta dos operadores *AND* e *OR*". As mutações aplicadas são similares as realizadas pelo operador de mutação *ClassEquivalentUndefAnd*, diferenciando apenas que ao invés de remover o operador *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.

4.2.24 Operador de mutação AxiomEquivalentDefNot - AEDN

- **Objetivo:** Adicionar uma definição de negação na equivalência entre classes.

- **Definição:** Dada uma classe C, tal que $C \in OWLClass$, existe relação do tipo *equivalentClass* entre C e outras classes do modelo, relação esta descrita na forma lógica (A *and* B), (A *or* B), (A *someValuesFrom* B) ou (A *allValuesFrom* B), sendo A e B classes do conjunto *OWLClass* ou outras expressões lógicas, então essa relação *equivalentClass* é alterada para (A *and (not* B*)*), (A *or (not* B*)*), (A *someValuesFrom (not* B*)*) ou (A *allValuesFrom (not* B*)*). A relação *equivalentClass* passa a ser descrita na forma lógica com o operador (*not*).

- **Descrição:** Quando uma classe é definida como equivalente por uma expressão lógica como (*A allValuesFrom B*), o operador *AxiomEquivalentDefNot* altera essa expressão lógica que define a equivalência da classe, adicionando o operador *not*.

- **Exemplo:** A classe *Atendimento* possui definição de equivalência *"Atendimento and ((hasPessoaEnvolvida allValuesFrom Profissional))"*. Essa definição seria alterada para *"Atendimento and ((hasPessoaEnvolvida allValuesFrom (not Profissional)))"* o que iria influenciar diretamente a classificação de indivíduos nesta classe.

O possível defeito revelado por este operador de mutação, é o "Uso incorreto da definição *some not*", pois ao aplicar a negação em uma definição de equivalência, este defeito pode ser revelado caso a negação definida seja de maneira extremamente genérica na definição da equivalência.

4.2.25 Operador de mutação AxiomEquivalentUndefNot - AEUN

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

- **Definição:** Dada uma classe *C*, tal que $C \in OWLClass$, existe relação do tipo *equivalentClass* entre *C* e outras classes do modelo, relação esta descrita na forma lógica (*A and (not B)*), (*A or (not B)*), (*A someValuesFrom (not B)*) ou (*A allValuesFrom (not B)*), sendo *A* e *B* classes do conjunto *OWLClass* ou outras expressões lógicas, então essa relação *equivalentClass* é alterada para (*A and B*), (*A or B*), (*A someValuesFrom B*) ou (*A allValuesFrom B*). A relação *equivalentClass* passa a ser descrita na forma lógica sem o operador (*not*).

- **Descrição:** Para uma classe definida como equivalente por uma expressão lógica como (*A allValuesFrom (not B)*), o operador *AxiomEquivalentUndefNot* altera essa expressão lógica que define a equivalência da classe, removendo o operador *not*.

- **Exemplo:** A classe *Atendimento* possui definição de equivalência *"Atendimento and ((hasPessoaEnvolvida allValuesFrom (not Profissional)))"*. Essa definição seria al-

terada para *"Atendimento and ((hasPessoaEnvolvida allValuesFrom Profissional))"* o que iria influenciar diretamente a classificação de indivíduos nesta classe.

Os possíveis defeitos revelados por este operador de mutação, são os mesmos apresentados na Seção 4.2.24, devido as mutações serem similares as realizadas pelo operador de mutação *AxiomEquivalentDefNot*.

4.3 Processos do teste de mutação

Para a aplicação do teste de mutação em ontologias OWL, tornam-se necessários as execuções dos seguintes processos: geração das ontologias mutantes com base nos operadores de mutação; geração dos dados de teste; execução da ontologia original com os dados de teste definidos; execução das ontologias mutantes com os mesmos dados de teste utilizados na ontologia original e análise dos resultados obtidos. Estes processos são detalhados nas seções 4.3.1 à 4.3.5.

4.3.1 Geração das ontologias mutantes

De acordo com Delamaro, Maldonado e Jino [20], é possível gerar mutantes com a aplicação de mais de um operador de mutação de uma só vez. Assim, ontologias mutantes geradas a partir de k alterações simultâneas na ontologia O, são chamadas de ontologias mutantes de ordem k.

Mutantes de ordens superiores possuem um custo de geração e execução extremamente alto, além de não contribuírem de forma significativa para a construção de casos de teste melhores [11]. Deste modo, nesta dissertação de mestrado, é utilizado apenas a geração de mutantes de primeira ordem.

O primeiro passo para aplicar o teste de mutação em uma ontologia OWL O, é gerar um conjunto de ontologias mutantes O'. Para isso, operadores de mutação devem ser selecionados, para que os desvios sintáticos mais comuns sejam modelados com base nos erros típicos cometidos pelos desenvolvedores.

A partir do grupo de operadores de mutação definido na seção 4.2, cada operador

é utilizado para realizar as alterações na ontologia O com base nas suas especificações, podendo o mesmo operador gerar um número arbitrário de ontologias O', com o mesmo tipo de defeito aplicado em pontos diferentes da ontologia O.

4.3.2 Geração dos dados de teste

Bons dados de teste podem ser definidos através das questões de competência (CQs), de modo que estas questões são estabelecidas em linguagens de consultas e expressam o domínio do conhecimento que a ontologia descreve, podendo serem definidas em *DL Query* e executadas através da ferramenta de auxílio *Protégé* [67].

Os dados de teste utilizados para a execução da ontologia O, devem ser aplicados com o auxílio de uma ferramenta que interprete as linguagens de consultas de ontologias, como *DL Query* ou *SPARQL*, a ferramenta deve dispor de um classificador para a checagem de consistência e inferência na hierarquia de classes, tal como a ferramenta *Protégé* [67].

4.3.3 Teste da ontologia original

Após a definição do conjunto de dados de teste T, deve ser executada a ontologia O com T. Cada dado de teste aplicado é interpretado por 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.

Após a aplicação de cada dado de teste deve-se analisar os resultados obtidos, de modo que caso sejam retornados resultados inesperados, fica a critério do testador definir se os resultados estão corretos, caso contrário, um erro foi detectado em O.

4.3.4 Teste das ontologias mutantes

Cada dado de teste T executado na ontologia O, deve ser executado em cada ontologia mutante O' gerada. Se a execução de um mutante O' com um dado de teste T apresenta um resultado diferente da execução de O com T, então T apresentou a diferença entre O e O', determinando que O' está morto. Caso contrário, define-se que O' revelou um defeito

em O, ou que O' é equivalente a O, de modo que também pode-se inferir que houve uma falha em T por não conseguir distinguir O de O', necessitando que novos dados de teste sejam gerados.

A definição dos mutantes como equivalentes fica a critério do testador, de modo que o mesmo deve identificar se existe a necessidade de gerar novos dados de teste, caso o conjunto utilizado não apresente resultados satisfatórios.

Após a execução de todos os mutantes, pode-se aplicar o escore de mutação para averiguar o grau de adequação dos casos de teste utilizados. Deste modo, o escore de mutação, conforme descrito na Seção 3.2, é adaptado nesta Seção de acordo com o número de ontologias mutantes geradas, mortas e definidas como equivalentes. Assim, dada uma ontologia O e o conjunto de casos de teste T, calcula-se o escore de mutação de acordo com o número de ontologias mutantes mortas pelo número total de mutantes gerados menos o número de mutantes definidos como equivalentes a O, conforme apresentado a seguir:

$$\text{ms(T)} = \frac{Om}{(Og - Oe)}, \text{ onde:}$$

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

De modo que quanto mais próximo de 1 resultar o escore de mutação, mais adequados estão os casos de teste utilizados.

4.3.5 Análise dos resultados

Na execução de cada caso de teste em O', toda vez que o resultado obtido é diferente do resultado de O, o mutante O' é definido como morto e descartado. Para cada O' que não apresente resultados diferentes de O com nenhum caso de teste T, e não sejam gerados novos casos de teste que produzam um resultado diferente, O' pode ter revelado um defeito existente em O, ou pode ser definido como equivalente a O.

Definir se uma ontologia mutante O' é equivalente ou não a O é uma questão indecídivel, assim, decidir se o teste continua ou não enquanto O' não apresenta um resultado que diferencie de O, fica a critério do testador, de modo que se o escore de mutação for 1, ou estiver bem próximo de 1, pode-se encerrar o teste e considerar o conjunto de casos de teste T utilizado adequado.

4.4 Considerações finais

Neste capítulo foram apresentados os métodos para a aplicação do teste de mutação para ontologias OWL, assim como foram propostos e apresentados os operadores de mutação para a realização das alterações sintáticas na ontologia em teste e geração das ontologias mutantes.

Para cada operador de mutação, foram associados os possíveis defeitos que os mesmos podem revelar, facilitando desta maneira a escolha dos operadores a serem utilizados de acordo com os defeitos que se deseja descobrir. Neste contexto, também é mostrado a definição de cada operador, assim como um exemplo prático de aplicação.

Dos 25 operadores de mutação definidos para o teste de mutação de ontologias OWL, 06 foram elaborados com base no trabalho de Lee, Bai e Chen [46], sendo os operadores CED, CEU, CECO, CDD, CDU e CEUO. Outros 04 operadores foram definidos com base no trabalho de Derezinska [23], sendo os operadores CUC, CUP, PID e PIU. Assim, os demais operadores apresentados foram elaborados conforme as características de desenvolvimento da linguagem OWL e dos defeitos apresentados.

Verifica-se com base na análise dos operadores definidos, que os mesmos simulam na maioria dos casos, erros de representação semântica das estruturas representadas nas ontologias, de modo que dos 25 operadores apresentados, 23 aplicam-se nas alterações em axiomas e somente 2 são utilizados para realizar mutações na hierarquia das classes. Assim, é possível validar a hipótese de que a maioria dos erros cometidos no desenvolvimento de ontologias OWL são erros semânticos, conforme é possível observar através da análise dos erros demonstrados no Capítulo 2.

CAPÍTULO 5

ESTUDO DE CASO

Para analisar detalhadamente a aplicação dos operadores de mutação propostos, assim como a execução dos processos para a aplicação do teste de mutação em ontologias OWL apresentados na Seção 4.3, foi realizado um estudo de caso no contexto de registros eletrônicos de saúde. Os conceitos do domínio clínico destes registros apresentados em arquétipos são convertidos para ontologias OWL, devido às dificuldades de implementação apresentadas pela linguagem ADL (do inglês, *Archetype Definition Language*), na qual os arquétipos são desenvolvidos [49].

5.1 Objetivos do estudo de caso

a) Aplicar o teste de mutação em ontologias OWL, para avaliar os operadores de mutação propostos;

b) Revelar defeitos ocasionados no processo de desenvolvimento de ontologias OWL, que não seriam revelados sem a execução do teste de mutação;

c) Analisar os resultados obtidos e definir o escore de mutação;

d) Validar o teste de mutação como uma técnica aplicável ao teste de ontologias OWL.

Para atender à estes objetivos, foram utilizados como base para esta pesquisa arquétipos desenvolvidos no padrão openEHR[1], de modo que após serem convertidos para ontologias OWL, aplicou-se o teste de mutação. Para auxiliar o processo de teste e aplicação dos operadores de mutação, foi utilizado a ferramenta *Protégé* [67], de criação e edição de ontologias. A motivação para a utilização de arquétipos como base para este trabalho é descrito na Seção 5.2.

[1]Para representação de registros eletrônicos de saúde também podem ser utilizados outros modelos como MLHIM [52], ISO13606 [40] e HL7 [37].

5.2 Embasamento teórico

Registro Eletrônico de Saúde (RES) é composto pelo registro de informação de saúde de um paciente, de modo que esse registro é elaborado a partir de eventos ocorridos em múltiplas organizações de saúde, sendo composto de um ou vários repositórios de dados clínicos e demográficos de pacientes [30]. O RES deveria disponibilizar a informação clínica semânticamente interoperável do paciente para além dos limites de uma organização.

Compreende-se a diversidade de estruturas de informações heterogêneas que compõem o registro eletrônico de saúde, de modo a serem muitos os desafios e mudanças culturais enfrentados no desenvolvimento de um sistema de RES [42]. Do ponto de vista técnico, o desafio da interoperabilidade e a natureza complexa e dinâmica das informações em saúde, fazem seu desenvolvimento ser mais árduo do que o de outros sistemas de informação [55].

Uma alternativa para o problema da interoperabilidade dos sistemas de RES, é projetar sistemas para o domínio clínico utilizando uma das propostas existentes para a modelagem multinível, como o modelo proposto pela fundação openEHR [36].

5.2.1 Modelagem Multinível openEHR

O modelo openEHR fornece um método de implementação de registros de conteúdo clínico através de um modelo de dois níveis [6, 51]:

- Modelo de Referência (do inglês, *Reference Model* - RM): Representa uma arquitetura lógica de informação, composto por classes que representam estruturas genéricas do registro eletrônico de saúde ou de dados demográficos, além de fornecer mecanismos para controle de versão, auditoria, controle de acesso aos dados, assim como a sintaxe básica de declarações.

- Modelo de Arquétipos (do inglês, *Archetype Model* - AM): Composto por especificações formais, de meta-nível, que restringem o conteúdo dos dados representados no RM. As alterações de estrutura e de regras de negócios refletem-se no AM e não no RM, de modo que não sejam necessárias alterações no mecanismo de persistência

utilizado. Além disso, os arquétipos devem ser criados e mantidos por especialistas do domínio, podendo ser associados a ontologias e incluir referências a sistemas terminológicos utilizados para codificar os seus dados.

O Modelo de Referência propõe um modelo de informação capaz de representar as características globais de um registro de saúde, permitindo a representação de diversos conceitos por meio dos arquétipos.

De acordo com Santos [24], as classes do modelo de referência baseiam-se na prática realizada por instituições e profissionais de saúde, sendo definidas na Figura 5.1.

COMPONENTES	DESCRIÇÃO
EHR_EXTRACT	Classe de maior nível da hierarquia de informação de um RES, utilizada para efeito de comunicação entre um sistema provedor (que fornece a informação) e um sistema destinatário (que a recebe).
FOLDER	Classe utilizada para o agrupamento e organização de *COMPOSITIONS* que se classificam em um mesmo tema ou assunto.
COMPOSITION	Classe que registra o conjunto de informações resultantes de um encontro clínico ou a sequência de dados de um formulário clínico.
SECTION	Classe utilizada para organização de dados dentro de uma *COMPOSITION*, usualmente refletindo o fluxo de informação realizado no encontro clínico ou estruturado para facilitar a compreensão de um usuário.
ENTRY	Classe que representa o resultado de uma ação, observação, interpretação, ou intenção clínica para registro em um RES.
CLUSTER	Classe de estruturas de dados múltiplas aninhadas, utilizadas para dados compostos como: séries de tempo, tabelas, etc..
ELEMENT	Classe que representa o último nível da hierarquia de informação de um RES, cujo conteudo é o dado propriamente dito.

Figura 5.1: Descrição das classes do modelo de referência openEHR [24]

O modelo de referência openEHR, apresentado a seguir na Figura 5.2, organiza os dados em COMPOSITIONs, que são responsáveis por registrar o conjunto de informações resultantes de um encontro clínico. Opcionalmente estas COMPOSITIONs podem ser incluídas dentro de uma estrutura hierárquica de FOLDERs, cuja finalidade é o agrupamento e organização de COMPOSITIONs. As COMPOSITIONs contém ENTRYs e GENERIC_ENTRYs, que podem ser incluídas dentro de uma estrutura hierárquica de SECTIONs, que possuem a finalidade de organizar os dados dentro de uma COMPOSITION [24, 49].

Toda informação criada em um Registro Eletrônico de Saúde openEHR é registrada como um subtipo de ENTRY, devendo ser definida como ADMIN_ENTRY, OBSERVATION, EVALUATION, INSTRUCTION ou ACTION. Qualquer tipo de ENTRY é composta por ELEMENTs, que são responsáveis por representar a estrutura final do dado sendo armazenado, opcionalmente sendo organizados através de uma hierarquia de CLUSTERs, de modo a melhorar a organização de dados compostos [49, 8].

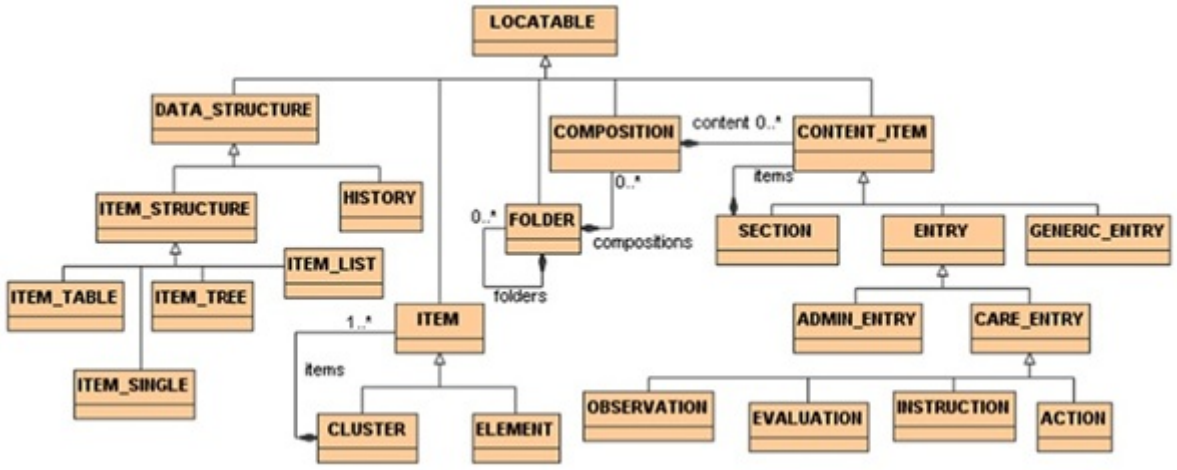

Figura 5.2: Modelo de Referência openEHR [49]

Para cada classe do modelo de referência, vários modelos de arquétipos podem ser desenvolvidos, representando todos os assuntos possíveis em um atendimento clínico.

5.2.2 Arquétipos openEHR

Arquétipos são definições de conceitos clínicos de um domínio específico na forma de combinações estruturadas e restrições ao Modelo de Referência. Podem ser considerados similares a um software escrito em uma linguagem de programação declarativa [7, 12].

Um arquétipo define o que deve ser armazenado em um repositório, sem definir procedimentos semânticos, em geral, requerem evidências e conhecimentos diversos, tais como: regras de aplicação, intervalos de medidas, tipos de dados, formatos de apresentação, melhor condição de representação dos dados (códigos, terminologias), ontologias, dados para suporte à tomada de decisão, restrições para integridade de dados e unidades de medida permitidas com intervalos numéricos associados [24].

O processo de desenvolvimento de um arquétipo consiste principalmente de traduzir um conceito clínico para entidades do Modelo de Referência, definindo a estrutura e representação dos dados em um Registro Eletrônico de Saúde [66]. As fontes potenciais de informação para o seu desenvolvimento são variadas: conhecimento de especialistas, publicações especializadas, telas de entrada de dados e relatórios de sistemas, listas de termos e tabelas usadas, modelos de dados de sistemas clínicos, mensagens e formulários regulatórios, formulários utilizados em consultas clínicas ou prontuários em papel [24].

Os arquétipos clínicos desenvolvidos conforme o modelo openEHR utilizam como padrão de desenvolvimento a Linguagem de Definição de Arquétipos - ADL. Esta linguagem apresenta uma sintaxe abstrata, que pode ser utilizada para expressar arquétipos baseados em qualquer modelo de informação, porém apresenta dificuldades para implementação de qualquer atividade semântica no conteúdo que esteja sendo representado [49].

5.2.3 Linguagem de Definição de Arquétipos (ADL)

ADL é uma linguagem formal para expressar arquétipos, de modo que documentos ADL são arquivos de texto estruturados, cuja estrutura é independente de qualquer norma ou domínio. De um modo geral, ADL não é uma linguagem para domínios clínicos, ela pode ser usada para a definição de qualquer tipo de arquétipo [50].

ADL é uma linguagem muito flexível, uma vez que a mesma estrutura pode ser utilizada para a especificação de arquétipos para diferentes modelos de referência, sendo assim, é capaz de representar a mesma estrutura sintática entre diversos modelos, porém apresenta dificuldades de implementação [50].

Um arquétipo ADL é composto basicamente de três partes [50, 44]: Seção HEADER, onde contém o nome do arquétipo, um código único que identifica o conceito clínico abordado, assim como informações de especialização e a linguagem em que foi escrito; a seção DEFINITION, especifica a estrutura e restrições associadas ao conceito clínico, essa estrutura restringe a cardinalidade e o conteúdo das instâncias do modelo de informação conforme definido pelo arquétipo, sendo esta a seção utilizada neste trabalho para a

aplicação do teste de mutação; e a seção ONTOLOGY, responsável por representar as definições terminológicas para cada conceito clínico sendo representado, e associações para outras terminologias. A Figura 5.3 apresenta um exemplo de modelo de arquétipo openEHR escrito em ADL.

```
archetype
        openEHR-EHR-OBSERVATION.pregnancy_test.v1
concept
        [at0000]
language
        original_language = <[ISO_639-1::en]>
definition
OBSERVATION[at0000] matches {
        data matches {
                HISTORY[at0001] matches {
                        events cardinality matches {1..*; unordered} matches {
                                POINT_EVENT[at0002] occurrences matches {0..1} matches {
                                        data matches {
                                                ITEM_SINGLE[at0003] matches {
                                                        item matches {
                                                                ELEMENT[at0004] matches {
                                                                        value matches {
                                                                DV_CODED_TEXT matches {
                                                                        defining_code matches {
                                                                                [local::
                                                                                at0005,
                                                                                at0006,
                                                                                at0007,
                                                                                at0008]
}}}}}}}}}}}}}
ontology
        term_definitions = <["en"] = <    items = <
        ["at0000"] = <
                text = <"Pregnancy test">
                description = <"Result of test using a commercial product to test for early pregnancy">
                >
        ["at0001"] = <
                text = <"Event Series">
                description = <"@ internal @">
                >
                        .
                        .
                        .
        ["at0007"] = <
                text = <"Weakly positive">
                description = <"The result is positive but the change is not strong">
                >
        ["at0008"] = <
                text = <"Strongly positive">
                description = <"The resulting change is very obvious">
>>>>
```

Figura 5.3: Exemplo de um arquétipo para exame de gravidez em ADL [57]

No exemplo acima, a seção HEADER inclui o nome do arquétipo, definido como *openEHR-EHR-OBSERVATION.pregnancy_test.v1*, podendo ser observado qual o modelo de referência em que esta sendo baseado (*openEHR-EHR*), qual o subtipo de ENTRY em que os dados clínicos são estruturados (OBSERVATION), o nome do conceito clínico (*pregnancy_test*) e, finalmente, o identificador da versão em que o arquétipo se encontra (*v1*), neste caso sendo a primeira versão lançada. Neste exemplo ainda é possível observar na seção HEADER o idioma em que o arquétipo foi escrito ([ISO_639-1::en]).

Na seção DEFINITION a estrutura do arquétipo é definida por uma OBSERVATION, onde os exames são representados como um histórico de eventos através de um HISTORY.

Cada evento ocorrido é representado através de um POINT_EVENT, onde os dados do resultado obtido são armazenados em uma estrutura de um único nível, demonstrada através de um ITEM_SINGLE composto por um ELEMENT do tipo DV_CODED_TEXT, podendo para cada evento armazenar somente um dos quatro valores estabelecidos (*at0005, at0006, at0007 e at0008*), os quais são associados a conceitos representados por uma terminologia externa.

A seção ONTOLOGY apresenta a definição de todos os conceitos representados pelo arquétipo, de modo que cada nó de arquétipo recebe um código único como *at0001*, sendo definidos na seção ONTOLOGY e muitas vezes relacionados a alguma terminologia externa como *LOINC* (do inglês, *Logical Observation Identifiers Names and Codes*) [39] ou *CID10* (Classificação Internacional de Doenças) [15].

5.2.4 Modelo de Objeto de Arquétipos (AOM)

Um arquétipo clínico ADL deve ser escrito para um modelo de referência específico [50] como openEHR, e obter objetos de um Modelo de Objetos de Arquétipos (do inglês, *Archetype Object Model* - AOM) abstrato. Desta forma, o processo de análise retorna uma coleção de objetos sintáticos e não semânticos [50]. A Figura 5.4 apresenta um exemplo das principais classes do AOM que estruturam um arquétipo.

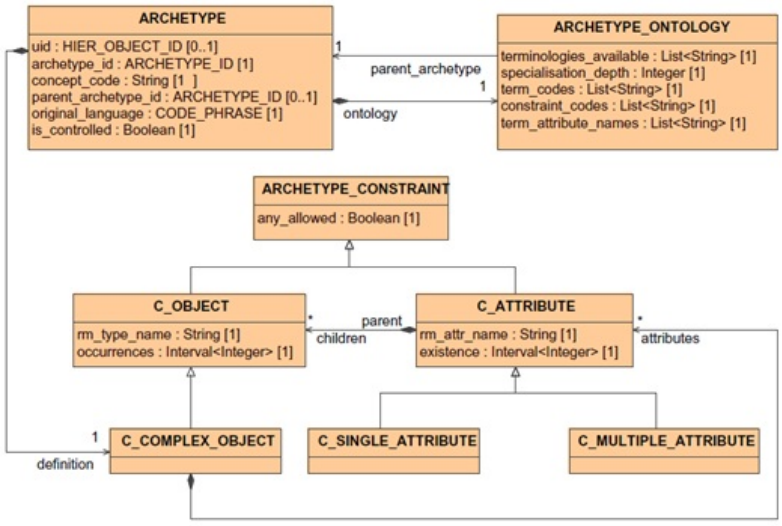

Figura 5.4: Modelo de Objetos de Arquétipos (AOM) [50]

Para realizar um processamento semântico do conteúdo de um arquétipo ADL, seriam necessários dois elementos: um *parser* ADL, neste trabalho definido como analisador ADL, para capturar objetos AOM, e um analisador do modelo de referência em particular para garantir a exatidão clínica do conteúdo [50].

Conforme um analisador ADL, o exemplo da Figura 5.3 seria analisado da seguinte maneira: seria definido como um *C_COMPLEX_OBJECT*, sendo composto por um conjunto de propriedades, entre as quais três são relevantes:

- (1) *rm_type_name : String*;
- (2) *node_id : String*;
- (3) *attributes* : conjunto de *C_ATTRIBUTE*.

A primeira propriedade determina a estrutura que está sendo representada conforme o modelo de referência (neste caso OBSERVATION); *node_id* representa a identificação de cada nó de arquétipo, como neste caso *at0000* e *attributes* referem-se as restrições definidas pelo modelo de referência representadas por *rm_type_name*.

A análise do arquétipo ADL da Figura 5.3 produziria cinco principais nós *C_COMPLEX_OBJECT*, tendo os seguintes valores para os triplos (*rm_type_name*, *node_id*, *attributes*):

- (1) (*"OBSERVATION", "at0000", data*);
- (2) (*"HISTORY", "at0001", events*);
- (3) (*"POINT_EVENT", "at0002", data*);
- (4) (*"ITEM_SINGLE", "at0003", item*);
- (5) (*"ELEMENT", "at0004", value*).

A representação gráfica do AOM deste arquétipo é mostrada na Figura 5.5. O analisador ADL produz um conjunto de objetos do AOM sem relações semânticas explícitas entre eles [50]. A semântica é desconhecida para o analisador e apenas a associação entre

os elementos das seções *definition* e *ontology* podem ser definidas. Assim, as possibilidades de raciocínio da ADL são atualmente muito limitadas, bem como a disponibilidade de ferramentas para usar e gerenciar conteúdos ADL são reduzidas. Consequentemente, a formalização de processos de troca e transformação de dados entre sistemas torna-se mais dificil do que utilizar modelos orientados à semântica, tais como ontologias.

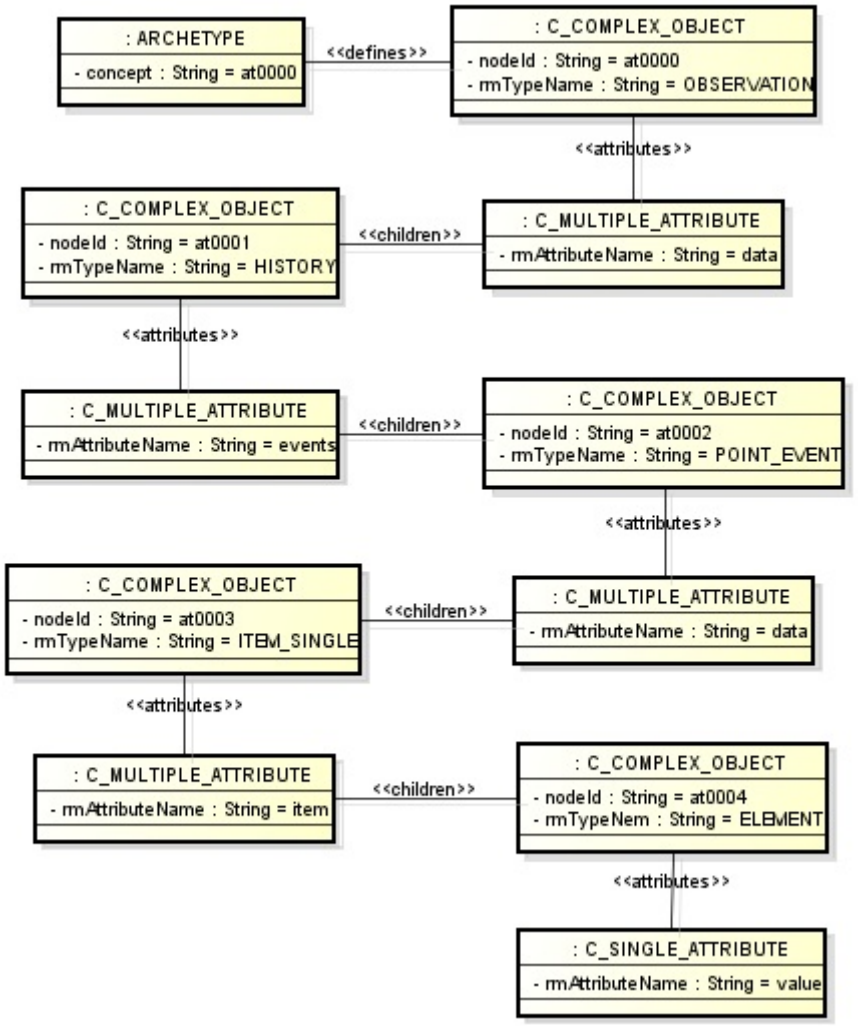

Figura 5.5: Extrato do Modelo de AOM para o arquétipo de exame de gravidez

Uma alternativa a estes problemas é a representação de arquétipos ADL em ontologias OWL. A OWL permite definir distinções significativas, detalhadas, precisas e consistentes entre classes, propriedades e relações definidas pelo usuário [50]. Deste modo, a construção de arquétipos baseados em OWL pode garantir a consistência do conhecimento, que não pode ser obtido através de técnicas ADL, devido a falta de ferramentas para gerenciar estes modelos.

Pesquisas em Web Semântica tem apresentado que as ontologias são fortes estruturas para a representação de tarefas de gerenciamento do conhecimento relacionadas a arquétipos e sistemas EHR [49], de modo que diversas metodologias e ferramentas para a comparação de diferentes ontologias, mesclando e identificando inconsistências, foram propostas. Assim, atividades como comparação, seleção, classificação e verificação de consistência podem ser realizadas sobre o conteúdo OWL de uma forma mais genérica, mais fácil e mais eficiente do que sobre o conteúdo ADL, já que de fato OWL é uma linguagem de representação de conhecimento para ambientes da Web Semântica.

O uso de ontologias para representar conhecimentos biomedicos não é um processo novo, visto que ontologias tem sido fortemente utilizadas em domínios biomedicos nos últimos anos em diferentes propósitos [64, 63]. Desta maneira, OWL torna-se apropriada para representar arquétipos clínicos e informações sobre registros eletrônicos de saúde, de modo que restrições sobre arquétipos podem ser determinadas por meio de restrições OWL ou pela definição de elementos apropriados.

Deste modo, neste estudo de caso é realizado a tradução de arquétipos ADL para ontologias OWL, devido aos problemas de representação semântica de informação da linguagem ADL, sendo aplicado em seguida o teste mutação nos arquétipos convertidos para ontologias OWL.

5.2.5 Conversão de ADL para OWL

A linguagem ADL apresenta uma sintaxe abstrata, que permite definir arquétipos para qualquer modelo de informação. Como estes arquétipos são usados para orientar a prática clínica, sendo necessário a exploração, comparação, classificação e integração de informações oriundas de diversos sistemas heterogêneos, a linguagem OWL apresenta excelentes mecanismos para estas atividades, possibilitando a representação de arquétipos ADL em ontologias OWL, proporcionando assim, a representatividade semântica dos conceitos clínicos abordados.

A representação de arquétipos em OWL exige a interpretação semântica do arquétipo clínico, de modo que as estruturas ADL definidas no arquétipo especializam as classes do

Modelo de Referência openEHR [44].

Deste modo, o primeiro passo para representar arquétipos ADL em OWL, é definir em OWL as classes do Modelo de Referência utilizadas pelo arquétipo. Um algoritmo para mapeamento de objetos ADL para OWL é apresentado por Elkin et al. [25].

1. As classes do Modelo de Referência utilizadas pelo arquétipo são definidas como classes OWL;

2. Cada conceito clínico é definido como uma subclasse da classe correspondente ao Modelo de Referência representado pelo arquétipo;

3. Os operadores ADL que definem associações entre as classes e componentes do arquétipo são definidos como Propriedades de Objetos em OWL, sendo definido como domínio destas propriedades a classe do arquétipo que faz referência à propriedade sendo representada, e como limite a classe ou conjunto de classes que representam os dados que devem ser associados à classe definida como domínio desta propriedade;

4. Os operadores que representam as estruturas de dados em ADL são definidos como Propriedades de Tipos de Dados em OWL, sendo definido como domínio destas propriedades a cardinalidade máxima ou mínima que um objeto associado a esta propriedade pode representar referente ao seu tipo de dado literal, e como limite o tipo de dado literal à que a propriedade em questão representa.

5. As restrições de propriedades e cardinalidades de relacionamentos estabelecidos em um arquétipo ADL, são convertidos em OWL através das restrições de cardinalidades possíveis pela linguagem.

Para o processo de conversão e representação de arquétipos ADL em ontologias OWL foi utilizado o *Protégé*, uma ferramenta para criação e edição de ontologias OWL, desenvolvido pelo centro de pesquisa de informática biomédica da Universidade de Stanford [67].

- **Definição das classes em OWL**

Basicamente um arquétipo possui uma estrutura hierarquica e restrições iniciadas com uma classe raiz. Assim, para um arquétipo ADL que apresente vários componentes do Modelo de Referência do mesmo tipo, como ELEMENTs por exemplo, em OWL é definido apenas uma classe para representar este componente como a classe raiz da hierarquia, e os demais são definidos como subclasses desta classe raiz para representarem os conceitos clínicos definidos no arquétipo. A Figura 5.6 apresenta uma parte de um arquétipo ADL, sendo que a Figura 5.7 representa a mesma estrutura de classes em OWL.

```
definition
        EVALUATION[at0000] matches {        -- Alert
                data matches {
                        ITEM_LIST[at0001] matches {   -- List
                                items cardinality matches {1..*; unordered} matches {
                                        ELEMENT[at0015] occurrences matches {0..1} matches{        -- Alert
                                                value matches {
                                                        DV_TEXT matches {*}
                                                }}
                                        ELEMENT[at0002] occurrences matches {0..1} matches {        -- Category
                                                value matches {
                                                        DV_TEXT matches {*}
                                                }}}}
                                                        .
                                                        .
                                                        .
ontology
        term_definitions = <
                ["de"] = <
                        items = <
                                ["at0000"] = <
                                        text = < "Alert">
                                        description = <"An assertion of potential threat to the welfare of the subject of care, carer or
                                                        healthcare provider."> >
                                ["at0001"] = <
                                        text = <"List">
                                        description = <"@ internal @"> >
                                ["at0002"] = <
                                        text = <"Category">
                                        description = <"The category of alert.">
                                        comment = <"Coding of the category of alert with a terminology is desirable, where possible.
                                                        For example, Medical Device Insertion; Significant Organism; or Social
                                                        Circumstance."> >
                                .
                                .
                                .
                                ["at0015"] = <
                                        text = <"Alert">
                                        description = <"Identification of the alert.">
                                        comment = <"For example, pacemaker, Methiciliin Resistant Staphylococcus Aureus(MRSA),
    or aggressive dog.">>
```

Figura 5.6: Arquétipo ADL representando o conceito de uma Ameaça Potencial [57]

Conforme o exemplo da Figura 5.6, EVALUATION, ITEM_LIST e ELEMENT re-

presentam as classes do Modelo de Referência openEHR utilizadas pelo arquétipo para restringir os conceitos clínicos que representam uma *Ameaça Potencial* a um sujeito.

Deste modo, são criadas três classes em OWL para representar cada uma destas estruturas do Modelo de Referência, de modo que cada uma delas será uma classe raiz para suas subclasses, conforme mostrado no exemplo da Figura 5.7.

Os códigos utilizados para representar cada nó de arquétipo, como *"at0000"*, *"at0001"*, são utilizados pela seção ONTOLOGY para associar cada nó de arquétipo a uma definição clínica, conforme alguma terminologia externa como LOINC ou CID10 por exemplo.

Assim, para a classe EVALUATION é criada uma subclasse denominada *Alert*, para a classe ITEM_LIST é definida uma subclasse *List*, e para a classe ELEMENT são atribuídas duas subclasses, *Category* e *Alert0* respectivamente.

```
<rdf:RDF xmlns="http://www.semanticweb.org/untitled-ontology-45#"
        <owl:Ontology rdf:about="http://www.semanticweb.org/untitled-ontology-45"/>

<owl:Class rdf:about="http://www.semanticweb.org/untitled-ontology-45#EVALUATION"/>
        <owl:Class rdf:about="http://www.semanticweb.org/untitled-ontology-45#Alert">
                <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/untitled-ontology-45#EVALUATION"/>
         </owl:Class>

<owl:Class rdf:about="http://www.semanticweb.org/untitled-ontology-45#ITEM_LIST"/>
        <owl:Class rdf:about="http://www.semanticweb.org/untitled-ontology-45#List">
                <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/untitled-ontology-45#ITEM_LIST"/>
        </owl:Class>

<owl:Class rdf:about="http://www.semanticweb.org/untitled-ontology-45#ELEMENT"/>
        <owl:Class rdf:about="http://www.semanticweb.org/untitled-ontology-45#Alert0">
                <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/untitled-ontology-45#ELEMENT"/>
         </owl:Class>

        <owl:Class rdf:about="http://www.semanticweb.org/untitled-ontology-45#Category">
                <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/untitled-ontology-45#ELEMENT"/>
         </owl:Class>

</rdf:RDF>
```

Figura 5.7: Classes OWL do conceito clínico Ameaça Potencial

Com base nas figuras 5.6 e 5.7, verifica-se que as classes EVALUATION, ITEM_LIST e ELEMENT, são classes do Modelo de Referência de Informação openEHR, e as classes *Alert*, *List*, *Alert0* e *Category* são classes que representam os conceitos clínicos definidos pelo arquétipo, conforme expostos pela seção ONTOLOGY.

- **Definição das Propriedades de Tipos de Dados**

As definições de propriedades dos dados de um arquétipo ADL devem ser representadas como Propriedades de Tipos de Dados em OWL, de modo que a comparação dos valores podem ser realizadas através da utilização dos construtores OWL *hasValue*, *someValuesFrom* e *allValuesFrom*.

Conforme o exemplo apresentado na Figura 5.6, deve ser criada uma Propriedade de Tipos de Dados OWL para cada um dos ELEMENTs representados pelo arquétipo, assim, conforme o exemplo, são definidas as propriedades *value_matches_alert* e *value_matches_category*, ambas com seus tipos de dados (*limite*) definidos como *string*, devido cada um dos *elements* serem representados como *dv_text*.

```
<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/untitled-ontology-45#value_matches_alert">
    <rdfs:range rdf:resource="&www;XMLSchema#string"/>
    <rdfs:domain>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://www.semanticweb.org/untitled-ontology-45#value_matches_alert"/>
        <owl:maxQualifiedCardinality rdf:datatype="&www;XMLSchema#nonNegativeInteger">1</owl:maxQualifiedCardinality>
        <owl:onDataRange rdf:resource="&www;XMLSchema#string"/>
      </owl:Restriction>
    </rdfs:domain>
  </owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/untitled-ontology-45#value_matches_category">
    <rdfs:range rdf:resource="&www;XMLSchema#string"/>
    <rdfs:domain>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://www.semanticweb.org/untitled-ontology-45#value_matches_category"/>
        <owl:maxQualifiedCardinality rdf:datatype="&www;XMLSchema#nonNegativeInteger">1</owl:maxQualifiedCardinality>
        <owl:onDataRange rdf:resource="&www;XMLSchema#string"/>
      </owl:Restriction>
    </rdfs:domain>
</owl:DatatypeProperty>
```

Figura 5.8: Representação OWL das propriedades de dados ADL

De acordo com o exemplo da Figura 5.6, cada *element* representado pelo arquétipo identifica no mínimo 0 e no máximo 1 ocorrência para cada registro, determinando que este componente não é requerido, deste modo, esta cardinalidade é representada em OWL como o domínio desta propriedade, através do construtor *maxQualifiedCardinality*, conforme apresentado na Figura 5.8.

- **Definição das Propriedades de Objetos**

Na hierarquia de um arquétipo ADL, os operadores que definem as associações entre os componentes representados no arquétipo, referem-se as classes do Modelo de Referência

de Informação, portanto eles associam os componentes das classes de um nível mais baixo, aos componentes das classes de um nível mais alto, definindo assim as classes e subclasses do arquétipo [44].

Para cada operador estabelecido no arquétipo ADL, deve-se criar uma Propriedade de Objeto em OWL, de modo que o domínio de cada propriedade criada deve ser a classe que o define no arquétipo ADL e, o limite desta propriedade será a classe ou conjunto de classes à que esse operador faz referência.

Analisando a imagem da Figura 5.6, verifica-se a definição de três operadores, que devem ser representados em OWL como Propriedades de Objetos denominadas como *data_matches*, cujo domínio deve ser específicado a classe *Alert* e como limite a classe ITEM_LIST, a propriedade *items_cardinality_matches*, tendo como domínio a classe ELEMENT e limite a classe *List* e, por último, a propriedade *value_matches*, com ambos domínio e limite definidos com a classe ELEMENT. A Figura 5.9 apresenta a definição destas propriedades em OWL.

```
<owl:ObjectProperty rdf:about="http://www.semanticweb.org/untitled-ontology-45#data_matches">
    <rdfs:domain rdf:resource="http://www.semanticweb.org/untitled-ontology-45#Alert"/>
    <rdfs:range rdf:resource="http://www.semanticweb.org/untitled-ontology-45#EVALUATION"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.semanticweb.org/untitled-ontology-45#items_cardinality_matches">
    <rdfs:range rdf:resource="http://www.semanticweb.org/untitled-ontology-45#ELEMENT"/>
    <rdfs:domain rdf:resource="http://www.semanticweb.org/untitled-ontology-45#List"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://www.semanticweb.org/untitled-ontology-45#value_matches">
    <rdfs:range rdf:resource="http://www.semanticweb.org/untitled-ontology-45#ELEMENT"/>
    <rdfs:domain rdf:resource="http://www.semanticweb.org/untitled-ontology-45#ELEMENT"/>
</owl:ObjectProperty>
```

Figura 5.9: Representação OWL dos operadores de associação ADL

5.3 Materiais e métodos

Com o objetivo de analisar a corretude dos arquétipos em OWL, averiguar a adequação dos operadores de mutação propostos para ontologias OWL e revelar defeitos que não seriam identificados sem a aplicação do teste de mutação, foram estabelecidos os seguintes passos:

1. Seleção manual dos arquétipos ADL openEHR no repositório online CKM [57].

Foram selecionados 07 arquétipos do CKM que no momento da seleção estavam em estado de revisão. Três arquétipos são do tipo *OBSERVATION*, respectivamente os arquétipos *"Apgar score"*, *"Autopsy examination"* e *"Fetal Heart Rate"*. Dois são do tipo *EVALUATION*, sendo os arquétipos *"Alert"* e *"A health oriented check list"*. Um é do tipo *INSTRUCTION*, sendo o arquétipo *"Informed Consent Request"* e um do tipo *ACTION*, sendo o arquétipo *"Follow up action"*. A codificação em ADL da seção DEFINITION desses arquétipos é apresentada no Apêndice B.

2. Conversão manual dos arquétipos ADL para ontologias OWL com base no algoritmo de mapeamento de ADL para OWL [25], sendo o processo realizado com o auxílio da ferramenta *Protégé* [67];

3. Geração manual dos arquétipos OWL mutantes com base nos operadores definidos para ontologias OWL, sendo os mesmos gerados com auxílio da ferramenta *Protégé*. Ilustrações dessas ontologias são encontradas no Apêndice C.

4. Geração manual de dados de teste com o apoio da ferramenta *Protégé*. Os dados de teste foram definidos através de consultas que expressam o domínio que a ontologia representa, escritas em *DL Query*;

5. Execução automática do arquétipo original OWL em teste na ferramenta *Protégé*;

6. Execução automática e individual dos arquétipos OWL mutantes na ferramenta *Protégé*;

7. Coleta e análise dos resultados.

5.4 Resultados

Com o objetivo de revelar defeitos sintáticos e semânticos nas ontologias, os operadores de mutação propostos neste trabalho foram aplicados na definição dos arquétipos OWL. Dos 25 operadores definidos para aplicar as modificações nas ontologias, 06 não foram utilizados neste experimento, sendo os operadores CUP, CDUO, CDUA, PIUD,

ACATS e AEUN. Tais operadores não foram aplicados devido aos arquétipos utilizados não possuírem definições em sua estrutura que justificassem as mutações realizadas por estes operadores.

A Tabela 5.1 apresenta os resultados obtidos após a aplicação dos 19 operadores de mutação nos 07 arquétipos openEHR selecionados e convertidos para OWL. São apresentados os totais de ontologias mutantes geradas para cada operador de mutação, onde foi possível gerar um total de 2000 ontologias mutantes, assim como os totais de mutantes que foram mortos através da aplicação dos dados de teste, os mutantes definidos como equivalentes e o número de defeitos revelados por operador, de modo que os defeitos revelados são referentes aos arquétipos openEHR representados em OWL e não em seu formato original ADL.

Para os mutantes que não foram mortos infere-se que os dados de teste utilizados não são eficientes, por não produzirem resultados diferentes da ontologia original, devendo serem gerados novos dados de teste. Por outro lado, estes mutantes podem ser definidos como equivalentes à ontologia original, ou caso contrário um defeito foi identificado. A definição por uma destas possibilidades fica a critério do testador, devido isto ser uma questão indecidível [27, 21, 10], conforme exposto na seção 4.3.5.

Para esta análise, os mutantes que não foram mortos por nenhum dado de teste gerado, foram definidos como equivalentes ou revelaram defeitos nas ontologias originais de acordo com o tipo de mutação realizada pelo operador. Por exemplo, o operador ACATO revelou 04 defeitos, pois a substituição do operador AND pelo operador OR em um axioma altera o conceito que está sendo representado, de modo que neste caso os mutantes que não foram mortos não podem ser definidos como equivalentes.

Também é apresentado o escore de mutação obtido para cada operador, podendo ser analisado o grau de adequação de cada um. Vale salientar que o escore de mutação é calculado nesta análise como a razão entre o número de mutantes mortos pelo número de mutantes gerados menos o número de mutantes equivalentes.

Os operadores de mutação PRD e PRUP apresentaram 0 (zero) mutantes mortos. Considera-se a hipótese de que os dados de teste utilizados não foram suficientes para

Tabela 5.1: Mutantes gerados para cada operador de mutação

Operadores	**Número de mutantes gerados**			**Defeitos**	**Escore**
	Gerados	**Mortos**	**Equivalentes**		
CUC	74	50	24	0	1
CDD	66	49	0	17	0,742
CDDO	249	246	0	03	0,987
CEU	102	90	0	12	0,882
CED	251	251	0	0	1
CECO	499	499	0	0	1
CEUA	15	06	0	09	0,4
CEUO	104	71	33	0	1
PDD	58	34	0	24	0,586
PDUP	26	24	0	02	0,923
PRD	68	00	68	0	1
PRUP	14	00	14	0	1
PDU	40	35	0	05	0,875
PRU	86	01	0	85	0,116
PID	104	104	0	0	1
ACOTA	65	49	0	16	0,753
ACATO	09	05	0	04	0,555
ACSTA	91	82	0	09	0,901
AEDN	79	78	0	01	0,987
TOTAL	**2000**	**1674**	**139**	**187**	**0,899**

identificar os defeitos inseridos por estes operadores. Isto provavelmente porque não foram instanciados objetos para as ontologias em teste. Assim, estes mutantes foram definidos como equivalentes.

Ao contrário dos operadores PRU e CEUA que obtiveram o escore de mutação abaixo de 0,5, assim como dos operadores PDD e ACATO que ficaram abaixo de 0,6, os demais operadores apresentaram resultados satisfatórios.

Os mutantes gerados com o operador ACATO que não foram mortos por nenhum caso de teste, revelaram defeitos na ontologia original, de modo que nenhum mutante foi definido como equivalente, sendo os defeitos revelados referentes a "utilização incorreta dos operadores AND e OR", conforme descrito na Seção 4.2.21. De um total de 09 mutantes gerados, 04 revelaram defeitos. A substituição do operador "AND" pelo operador "OR" em um axioma, afeta diretamente a instanciação dos indivíduos associados ao axioma mutado, o que fez com que os defeitos fossem revelados.

Similar ao operador ACATO, o operador de mutação ACOTA realiza a substituição do operador "OR" pelo operador "AND" na definição de um axioma, afetando diretamente na instanciação dos indivíduos para as classes associadas ao axioma mutado. Deste modo, dos 65 mutantes gerados com este operador, foram revelados 16 defeitos nos modelos testados, não sendo definido nenhum mutante como equivalente. Também similar ao operador ACATO, os defeitos revelados referem-se a "utilização incorreta dos operadores AND e OR", conforme descrito na Seção 4.2.20.

A substituição da restrição *someValuesFrom* pela restrição *allValuesFrom* em um axioma realizada com o operador ACSTA, passou a determinar que somente seriam instanciados para as classes associadas ao axioma os indivíduos que são instâncias das classes referenciadas pela restrição *allValuesFrom*. Sendo assim, dos 91 mutantes gerados com este operador, foram revelados 09 defeitos nos modelos testados, de modo que nenhum mutante foi definido como equivalente, sendo os defeitos revelados referentes ao "uso equivocado de qualificadores de restrição" e "uso incorreto da restrição *AllValuesFrom*", conforme descrito na Seção 4.2.22.

A mutação realizada pelo operador AEDN afeta diretamente a instanciação dos indivíduos, de modo que a definição de uma negação em um axioma invalida que indivíduos da classe associada ao axioma mutado sejam instanciados. Assim, foi possível revelar 01 defeito em um total de 79 mutantes gerados com este operador, sendo o defeito encontrado referente ao "uso incorreto da definição *some not*", conforme a Seção 4.2.24.

A definição de disjunção entre classes realizada pelo operador CDD invalida que um indivíduo seja instanciado para ambas as classes ao mesmo tempo. Neste estudo de caso, dos 66 mutantes gerados com este operador foram mortos 49 com os dados de teste utilizados, sendo revelados 17 defeitos referentes a "classificação incompleta", "falta de disjunção" e "informações básicas incompletas", conforme apresentado na Seção 4.2.3 na Tabela 4.2.

Diferentemente do operador CDD, o operador de mutação CDDO aplica a disjunção entre classses a somente duas subclasses de uma superclasse. De um total de 249 mutantes gerados com este operador, foram revelados 03 defeitos similares aos apresentados pelo

operador CDD, sendo referentes a “falta de disjunção”, conforme apresentado na Seção 4.2.4, sendo que 246 mutantes foram mortos com os dados de teste utilizados, não sendo definido nenhum como equivalente.

Todos os mutantes gerados com o operador CECO foram mortos com os dados de teste utilizados, de modo que nenhum defeito foi revelado e também nenhum mutante foi definido como equivalente.

Foram gerados 251 mutantes com o operador CED, sendo todos mortos com os dados de teste utilizados, o que não revelou nenhum defeito na ontologia original e também não apresentou mutantes equivalentes.

A remoção de equivalência de uma classe com o operador CEU implica na falta de definições de equivalência entre propriedades e incompletude de informações da ontologia, o que possibilitou serem revelados 12 defeitos em um total de 102 mutantes gerados, referentes à “criação de elementos polissêmicos” e “uso incorreto de classes primitivas e definidas”, conforme exposto na Seção 4.2.8.

A remoção do operador “AND” de um axioma com o operador CEUA implica diretamente na instanciação dos indivíduos para a classe associada ao axioma mutado, o que conforme os operadores ACATO e ACOTA, revelou-se um alto número de defeitos referente ao total de mutantes gerados, sendo estes defeitos também referentes ao “uso incorreto dos operadores AND e OR” e o “uso incorreto de classes primitivas e definidas” como apresentado na Seção 4.2.10.

Similar ao operador CEUA, a remoção do operador “OR” de um axioma através do operador de mutação CEUO também implica diretamente na instanciação dos indivíduos para a classe associada ao axioma mutado, mas como o operador “OR” necessita que somente uma proposição do axioma seja válida para a instanciação dos indivíduos, todos os mutantes que não foram mortos foram definidos como equivalentes.

A modificação da hierarquia de uma ontologia com o operador CUC pode representar em primeiro momento a ocorrência de um defeito, mas devido aos axiomas definidos para as classes mutadas, o resultado após a execução da ontologia pode não revelar este defeito, de modo que todos os mutantes que não foram mortos com este operador foram definidos

como equivalentes.

A alteração de um domínio de uma propriedade com o operador PDD para um nível abaixo na hierarquia de classes, afeta diretamente na instanciação dos indivíduos para as classes associadas ao axioma mutado, podendo também representar uma definição equivocada. Do total de 58 mutantes gerados foram revelados 24 defeitos nos modelos em teste referentes ao "domínio e limite de propriedades" e "intervalo de dados ou domínio limitado", como descrito na Seção 4.2.12, não sendo definido nenhum como equivalente.

A definição de inversão entre duas propriedades com o operador PID, define que os indivíduos associados por uma propriedade sejam também associados de modo contrário através da outra propriedade. Foi possível gerar 104 mutantes com este operador, sendo que todos foram mortos com a aplicação dos dados de teste utilizados.

De modo semelhante ao operador PDD, a substituição do intervalo de uma propriedade para um nível abaixo na hierarquia de classes com o operador PRD também afeta diretamente a instanciação dos objetos para as classes associadas a esta propriedade, podendo também representar uma definição equivocada. Foram gerados ao todo 68 mutantes com este operador, porém, por não haver objetos instanciados para a ontologia neste estudo de caso, todos os mutantes foram definidos como equivalentes.

A remoção do intervalo de uma propriedade com o operador PRU possibilita que qualquer objeto seja instanciado para as classes associadas ao axioma mutado. Assim, de um total de 86 mutantes gerados, somente 01 foi morto com os dados de teste utilizados, de modo que os outros 85 possibilitaram a representação de defeitos nas ontologias em teste referentes à "falta de domínio ou intervalo em propriedades", "intervalo de dados ou domínio limitado" e "assumir características implícitas nos nomes dos componentes", como apresentado na Seção 4.2.17. Estes resultados também poderiam variar caso a ontologia em teste possuísse objetos instanciados.

Assim como o operador PRD, a alteração do intervalo de uma propriedade para um nível acima na hierarquia de classes com o operador PRUP, também afeta diretamente a instanciação dos objetos para as classes associadas a esta propriedade, podendo também representar uma definição equivocada. Assim, como não foi possível matar os mutan-

tes gerados com os dados de teste utilizados, pela ausência de objetos instanciados na ontologia, todos foram mantidos como equivalentes neste estudo de caso.

Similar ao operador PRUP, a alteração do domínio de uma propriedade para um nível acima na hierarquia de classes com o operador PDUP, afeta diretamente a instanciação dos objetos para as classes associadas a esta propriedade, podendo também representar uma definição equivocada. Este operador de mutação, ao contrário do operador PRUP onde não foi possível matar nenhum mutante pela ausência de objetos instanciados na ontologia, possibilitou revelar 02 defeitos a partir de um total de 26 mutantes gerados referentes ao "intervalo de dados ou domínio limitado" e "domínio e limite de propriedades", conforme a Seção 4.2.13.

A remoção do domínio de uma propriedade com o operador PDU, também afeta diretamente na instanciação dos indivíduos para as classes associadas a esta propriedade, deste modo, a partir de um total de 40 mutantes gerados com este operador, foi possível revelar 05 defeitos nas ontologias em teste referentes a "falta de domínio ou intervalo de dados", "intervalo de dados ou domínio limitado" e "assumir características implícitas nos nomes dos componentes", conforme descrito na Seção 4.2.16.

Os resultados dos testes individuais de cada arquétipo são apresentados nas tabelas no Apêncide A conforme a Tabela A.1, Tabela A.2 e a Tabela A.3, onde é possível observar a quantidade de mutantes gerados por operador de mutação em cada arquétipo, assim como a adequação de cada um em cada modelo testado.

É possível observar com os resultados deste estudo de caso, a grande quantidade de mutantes gerados e o alto escore de mutação dos operadores, assim como a existência de defeitos nos modelos OWL em teste, permitindo validar e adequar o modelo proposto de teste de mutação para ontologias. Alguns dos operadores utilizados revelaram defeitos que apresentaram falhas na execução da ontologia, retornando resultados insatisfatórios, mas que não foi possível matar os mutantes com os dados de teste utilizados. Com base nos operadores e testes aplicados, percebe-se a necessidade da aplicação de testes para ontologias.

5.5 Considerações Finais

Neste capítulo é apresentada a aplicação do teste de mutação para ontologias OWL, de modo que neste estudo de caso foram utilizados para efeito de análise dos operadores de mutação propostos 07 arquétipos clínicos openEHR, que foram convertidos para ontologias OWL.

Dos 25 operadores de mutação definidos, 19 foram aplicados nos modelos selecionados, devido os arquétipos não possuírem características que justificassem a aplicação dos demais operadores. É possível observar nos resultados obtidos na Tabela 5.1, o alto número de mutantes gerados, assim como o alto escore de mutação obtido na revelação dos defeitos através dos dados de teste utilizados.

Para os operadores de mutação que apresentaram 0 (zero) mutantes mortos, considera-se a hipótese de que os dados de teste utilizados não foram suficientes para identificar os defeitos inseridos por estes operadores. Isto provavelmente porque não foram instanciados objetos para as ontologias em teste.

Ao contrário dos operadores PRU e CEUA que obtiveram o escore de mutação abaixo de 0,5, assim como dos operadores PDD e ACATO que ficaram abaixo de 0,6, os demais operadores apresentaram resultados satisfatórios. Possivelmente em uma nova análise com a instanciação de objetos na ontologia em teste, estes operadores apresentarão resultados melhores em relação a este primeiro estudo de caso.

Em alguns casos não foi possível identificar o defeito aplicado pelo operador de mutação com nenhum dado de teste, devido não ter sido considerado nesta análise a instanciação de objetos para as classes dos arquétipos representados em OWL, o que possivelmente produziria resultados melhores. Por outro lado, alguns operadores obtiveram 100% de escore de mutação, sendo possível identificar todos os defeitos simulados pelos mesmos.

Com a aplicação do teste de mutação para ontologias OWL, foi possível comprovar a necessidade e eficácia da aplicação do método de teste para ontologias, assim como dos operadores de mutação propostos, pois mesmo obtendo-se uma adequação baixa em alguns operadores, o resultado final obtido entre todos os modelos testados, manteve-se em mais de 89% na identificação dos defeitos simulados.

CAPÍTULO 6

CONCLUSÃO

Ontologias definem conceitos, classes, propriedades, relações, restrições e axiomas sobre um determinado domínio do conhecimento, que podem representar o mundo real e conceitual através de identificadores semânticos. As ontologias demonstraram ser extremamente úteis no auxílio ao desenvolvimento de sistemas computacionais, devido algumas características próprias como um vocabulário para representação do conhecimento e a possibilidade de estender um modelo genérico para um domínio específico.

Um domínio de conhecimento específico pode ser modelado em ontologias de diversos modos, possibilitando que erros sejam cometidos e que defeitos inseridos possam causar falhas. Vários defeitos em ontologias foram expostos previamente neste trabalho [32, 62, 58, 13]. Alguns métodos de teste foram propostos por García-Ramos, Otero e Fernández-López [29] e por Vrandecic e Gangemi [68], entretanto estes testes apenas possibilitam analisar a validade da ontologia e não garantem que defeitos sejam revelados.

Foi proposto neste trabalho aplicar o teste de mutação para programas para a realização de testes em ontologias OWL. Foram definidos 25 operadores de mutação com base na grande quantidade de defeitos que podem ser cometidos no desenvolvimento de ontologias, e nas características de desenvolvimento da linguagem OWL.

Para validar os operadores de mutação propostos, foi realizado um estudo de caso no contexto de registros eletrônicos de saúde, onde arquétipos clínicos openEHR disponibilizados no repositório online CKM [57], foram convertidos para OWL e submetidos ao processo de teste. O método de teste pode ser aplicado a qualquer modelo de dados, não se restringindo somente ao domínio de registros eletrônicos de saúde.

Foi utilizado para auxiliar neste processo a ferramenta de desenvolvimento de ontologias *Protégé* [67], devido a falta de uma ferramenta para automatizar a aplicação dos operadores de mutação em ontologias OWL e geração dos mutantes.

Para a realização da conversão de um arquétipo openEHR para uma ontologia OWL, também foi utilizada esta ferramenta, de modo que após a geração da ontologia original, para cada mutação realizada com cada operador era gerada uma nova ontologia definida como ontologia mutante. Para a geração e execução dos dados de teste, foi utilizada a linguagem *DL Query*, uma linguagem de consultas para ontologias também disponibilizada na ferramenta *Protégé*.

Foram utilizados para este estudo de caso 07 arquétipos openEHR, sendo gerados em média 285 mutantes para cada modelo através de 19 operadores de mutação, de modo que 06 operadores não foram utilizados nesta análise devido a esses arquétipos não possuírem características que justificassem as aplicações.

Do total de 2000 ontologias mutantes obtidas com estes 19 operadores de mutação, foi possível matar 1674 mutantes com os dados de teste gerados, sendo que do total de mutantes que não foram mortos com nenhum dado de teste, 139 foram definidos como equivalentes às ontologias originais em teste, revelando-se assim um total de 187 defeitos nos 07 modelos testados. Foi possível obter um escore de mutação de mais de 89%, o que mostrou a eficácia dos operadores de mutação e dados de teste gerados, identificando os defeitos inseridos pelos operadores e revelando defeitos ocasionados pelo desenvolvedor.

O método proposto de teste mostrou-se eficaz para determinar se as ontologias desenvolvidas estão corretas, sendo possível gerar um alto número de mutantes e um alto escore de mutação, podendo-se analisar os possíveis defeitos cometidos pelo desenvolvedor.

6.1 Trabalhos futuros

- Realizar experimentos com outras ontologias de outros domínios de conhecimento;
- Implementar uma ferramenta que automatize a aplicação dos operadores de mutação para ontologias OWL, de modo que as ontologias mutantes sejam geradas automaticamente, assim como para a aplicação dos dados de teste necessários;
- Realização de novos testes com a ferramenta automatizada, por exemplo, com a utilização de objetos instanciados para as classes da ontologia. Isto permite ter uma

nova percepção da adequação dos operadores de mutação propostos e, a aplicação dos 06 operadores que não foram utilizados neste estudo de caso;

- Implementar o Modelo de Referência openEHR em OWL;
- Elaborar um método de validação da instância de dados de acordo com o modelo de domínio em OWL.

BIBLIOGRAFIA

[1] OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/, fevereiro de 2004.

[2] OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/, fevereiro de 2004.

[3] Hiralal Agrawal, Richard Allan DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, Edward W. Krauser, Rhonda. J. Martin, Aditya P. Mathur, e Eugene Spafford. Design of Mutant Operators for the C Programming Language. Relatório Técnico SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, março de 1989.

[4] Anneliese Andrews, Robert France, Sudipto Ghosh, e Gerald Craig. Test Adequacy Criteria for UML Design Models. *Journal of Software Testing, Verification and Reliability*, 13:95–127, 2003.

[5] Carlos Julian Menezes Araújo e Robson do Nascimento Fidalgo. Metamodelo para Banco de Dados.

[6] Sistine Ann Barretto, Jim Warren, e Andrrew Goodchild. Designing guideline-based workflow-enabled electronic health records. *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, volume 6, páginas 10 pp.–, janeiro de 2004.

[7] Thomas Beale. Archetypes Constraint-based Domain Models for Futureproof Information Systems. *OOPSLA 2002 workshop on behavioural semantics*, 2002.

[8] Thomas Beale, Dipak Kalra, Sam Heard, e David Lloyd. The openEHR Reference Model: EHR Information Model, agosto de 2008.

[9] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, e Lynn Andrea Stein. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/, fevereiro de 2004.

[10] Timothy A. Budd. *Computer Program Testing: Proceedings of the Summer School on Computer Program Testing held at SOGESTA, Urbino, Italy, June 29-July 3, 1981*, capítulo Mutation Analysis: Ideas, Example, Problems and Prospects, páginas 129–148. North-Holand Publishing Company, Amsterdam, June-July de 1981.

[11] Timothy A. Budd, Richard Allan DeMillo, Richard Jay Lipton, e Frederick G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, GIT-ICS, páginas 220–233, New York, NY, USA, 1980. Defense Technical Information Center.

[12] Rong Chen, Sebastian Garde, Thomas Beale, Mikael Nyström, Daniel Karlsson, Gunnar O Klein, e Hans Åhlfeldt. An Archetype-based Testing Framework. Stig Kjaer Andersen, Gunnar O. Klein, Stefan Schultz, Jos Aarts, e M. Cristina Mazzoleni, editors, *EHealth Beyond the Horizon: Get IT There*, volume 136 of *Studies in health technology and informatics*, páginas 401–406, Amsterdam, 2008. IOS Press.

[13] Oscar Corcho, Catherine Roussey, e Luis Manuel Vilches Blazquez. Catalogue of Anti-Patterns for formal Ontology debugging. *Atelier Construction d ontologies : vers un guide des bonnes pratiques, AFIA 2009*, maio de 2009.

[14] Simone das Graças Domingues Prado. *Um experimento no uso de ontologias para reforço da aprendizagem em educação a distância*. Tese de Doutorado, Universidade de São Paulo, 2004.

[15] DATASUS. CID10. `http://www.datasus.gov.br/cid10/V2008/cid10.htm`. Accessed: 2013-10-13.

[16] John Davies, Dieter Fensel, e Frank van Harmelen, editors. *Towards the Semantic Web: Ontology-driven Knowledge Management*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[17] Aurélio Burque de Holanda Ferreira. Dicionário do Aurélio. `http://www.dicionariodoaurelio.com/`. Accessed: 2014-03-10.

[18] Júnio César de Lima e Cedric L. de Carvalho. Ontologias - OWL (Web Ontology Language). Relatório técnico, Universidade Federal de Goiás, junho de 2005.

[19] Lirisnei Gomes de Souza e Jair C. Leite. Geração automática de dicionários explicativos em sistemas de informaçoes geográficas usando ontologia. *Revista Eletônica de Iniciação Científica*, 5:1–12, 2005.

[20] Márcio Eduardo Delamaro, José Carlos Maldonado, e Mario Jino. *Introdução ao Teste de Software*, volume 394 of *Campus*. Elsevier, 2007.

[21] Richard Allan DeMillo, Richard Jay Lipton, e Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, abril de 1978.

[22] Kathrin Dentler, Ronald Cornet, Annette ten Teije, e Nicolette de Keizer. Comparison of Reasoners for Large Ontologies in the OWL 2 EL Profile. *Semantic web*, 2(2):71–87, abril de 2011.

[23] Anna Derezinska. Object-Oriented Mutation to Asses the Quality of Tests. *Proceedings of 29th Euromicro Conference*, páginas 417–420. IEEE Computer Society, setembro de 2003.

[24] Marcelo Rodrigues dos Santos. *Sistema de registro eletrônico de saúde baseado na norma ISO 13606: aplicações na Secretaria de Estado de Saúde de Minas Gerais*. Tese de Doutorado, Universidade Federal de Minas Gerais, Minas Gerais, 2011.

[25] Peter L. Elkin, Martin Kernberg, Thomas Beale, Sam Heard, Mark Shafarman, Robert Dolin, e Dale Nelson. HL7 Template and Archetype Architecture. Template Special Interest Group, novembro de 2003.

[26] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, T. Sugeta, e Paulo Cesar Masiero. Mutation Testing Applied to Validate Specifications Based on Statecharts. *ISSRE – International Symposium on Software Reliability Systems*, páginas 210–219, novembro de 1999.

[27] Ledyvânia Franzotte. Utilizando análise de mutantes para realizar o teste de documentos XML Schema. Dissertação de Mestrado, Universidade Federal do Paraná, Curitiba, 2006.

[28] Frederico Freitas e Stefan Schulz. Ontologias, Web Semântica e Saúde. *Revista Eletrônica de Comunicação Informação & Inovação em Saúde*, 3(1), 2009.

[29] Sara García-Ramos, Abraham Otero, e Mariano Fernández-López. OntologyTest: A Tool Evaluate Ontologies through Tests Defined by the User. *Springer*, páginas 91–98, 2009.

[30] Dave Garets e Make Davis. Electronic Medical Records vs. Electronic Health Records: Yes, There Is a Difference. *HIMSS Analytics*, 2006.

[31] Dragan Gasevic, Dragan Djuric, e Vladan Devedzic. *Model Driven Engineering and Ontology Development*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[32] Asunción Gómez-Perez. Ontology Evaluation. *Springer*, páginas 251–274, 2004.

[33] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, junho de 1993.

[34] Thomas R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies - Special issue:*

the role of formal ontology in the information technology, 43(5-6):907–928, dezembro de 1995.

[35] Nicola Guarino. Formal Ontology, Conceptual Analysis and Knowledge Representation. *International Journal of Human-Computer Studies - Special issue: the role of formal ontology in the information technology*, 43(5-6):625–640, dezembro de 1995.

[36] Leslie Heather. openEHR - the World's Record. *PulseIT*, 6(6):50–55, novembro de 2007.

[37] HL7. Health Level Seven International. `http://www.hl7.org/`. Accessed: 2013-10-25.

[38] Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, e Chris Wroe. A Practical Guide To Building OWL Ontologies Using The Protege-OWL Plugin and CO-ODE Tools Edition 1.0. agosto de 2004.

[39] Regenstrief Institute Inc. LOINC. `http://loinc.org/`. Accessed: 2013-10-13.

[40] ISO13606. EN13606 Association. `http://www.en13606.org/`. Accessed: 2013-10-25.

[41] Thomas C. Jepsen. Just What Is an Ontology, Anyway? *IT Professional*, 11(5):22–27, 2009.

[42] Dipak Kalra. *Clinical Foundations and Information Architecture for the Implementation of a Federated Health Record Service*. Tese de Doutorado, University College London, London, 2002.

[43] Nilesh Kawane. Fault Detection Effectiveness of UML Design Model Test Adequacy Criteria. *14th International Symposium on Software Reliability Engineering*, páginas 17–20, 2003.

[44] Ozgur kilic, Veli Bicer, e Asuman Dogac. Mapping Archetypes to OWL. *Software Research and Development Center*, 2005.

[45] Jan Korst, Gijs Geleijnse, Nick Jong, e Michael Verschoor. Ontology-Based Information Extraction from the World Wide Web. Wim Verhaegh, Emile Aarts, e Jan Korst, editors, *Intelligent Algorithms in Ambient and Biomedical Computing*, volume 7, páginas 149–167. Springer Netherlands, 2006.

[46] Shufang Lee, Xiaoying Bai, e Yinong Chen. Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. *Simulation Symposium, Annual*, 0:149–156, 2008.

[47] Kheronn Khennedy Machado. Composição dinâmica de serviços web utilizando ontologias na descrição e planejadores hierárquicos em inteligência artificial. Dissertação de Mestrado, Universidade Federal do Paraná, 2010.

[48] Alexander Maedche e Steffen Staab. Ontology Learning for the Semantic Web. *IEEE Intelligent Systems*, 16(2):72–79, março de 2001.

[49] Catalina Martínez-Costa, Marcos Menárguez-Tortosa, e Jesualdo Tomás Fernández-Breis. An Approach for the Semantic Interoperability of ISO EN 13606 and OpenEHR Archetypes. *J. of Biomedical Informatics*, 43(5):736–746, outubro de 2010.

[50] Catalina Martínez-Costa, Marcos Menárguez-Tortosa, Jesualdo Tomás Fernández-Breis, e José Alberto Maldonado. A model-driven approach for representing clinical archetypes for Semantic Web environments. *Journal of Biomedical Informatics*, 42(1):150–164, fevereiro de 2009.

[51] Leonardo Lezcano Matiaz. *Combining Ontologies with Clinical Archetypes*. Tese de Doutorado, University of Alcalá, Alcalá, 2011.

[52] MLHIM. Multi-Level Healthcare Information Modelling. `http://mlhim.org/`. Accessed: 2013-10-25.

[53] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[54] Glenford J. Myers e Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[55] Fabiane Bizinella Nardon. Utilizando XML para Representação de Informação em Saúde. 2002.

[56] Fabiane Bizinella Nardon. *Compartilhamento de Conhecimento em Saúde utilizando Ontologias e Bancos de Dados Dedutivos*. Tese de Doutorado, Universidade de São Paulo, 2003.

[57] openEHR. openEHR Clinical Knowledge Manager. `http://www.openehr.org/ckm/`. Accessed: 2012-06-12.

[58] María Poveda-Villalón, Mari Carmen Suárez-Figueroa, e Asunción Gómez-Pérez. A Double Classification of Common Pitfalls in Ontologies. *Workshop on Ontology Quality (OntoQual 2010), Co-located with EKAW 2010*, outubro de 2010.

[59] Roger S. Pressman. *Engenharia de Software*. Makron Books do Brasil, São Paulo, 3 edition, 1995.

[60] DL Query. DL Query. `http://protegewiki.stanford.edu/wiki/DLQueryTab`. Accessed: 2013-09-22.

[61] Sandra Rapps e Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, abril de 1985.

[62] Alan Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, e Chris Wroe. OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. *Information Management Group / Bio Health Informatics Forum*, 2004.

[63] Stefan Schulz e U. Hahn. Part-whole representation and reasoning in formal biomedical ontologies. *Artificial Intelligence in Medicine*, 34:179–200+, 2005.

[64] Barry Smith. From Concepts to Clinical Reality: An Essay on the Benchmarking of Biomedical Terminologies. *Journal of Biomedical Informatics*, 39(3):288–298, junho de 2006.

[65] Samir Tartir, Ismailcem Budak Arpinar, e Amith P. Sheth. Ontological Evaluation and Validation. *Theory and Applications of Ontology: Computer Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[66] Marcos Menárguez Tortosa e Jesualdo Tomás Fernández-Breis. OWL-based reasoning methods for validating archetypes. *Journal of Biomedical Informatics*, 46(2):304–317, 2013.

[67] Stanford University. PROTÉGÉ Ontology Editor. `http://protege.stanford.edu/`. Accessed: 2013-06-15.

[68] Denny Vrandečić e Aldo Gangemi. Unit tests for ontologies. Mustafa Jarrar, Claude Ostyn, Werner Ceusters, e Andreas Persidis, editors, *Proceedings of the 1st International Workshop on Ontology content and evaluation in Enterprise*, LNCS, 2006.

[69] Denny Vrandečić, Johanna Völker, Peter Haase, Duc Thanh Tran, e Philipp Cimiano. A Metamodel for Annotations of Ontology Elements in OWL DL. York Sure, Saartje Brockmans, e Jürgen Jung, editors, *Proceedings of the Second Workshop on Ontologies and Meta-Modeling*, Karlsruhe, Germany, outubro de 2006. GI Gesellschaft für Informatik.

[70] Daya C. Wimalasuriya e Dejing Dou. Using Multiple Ontologies in Information Extraction. *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, páginas 235–244, New York, NY, USA, 2009. ACM.

[71] W. Eric Wong, Aditya P. Mathur, e José C. Maldonado. Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness. *Software Quality and Productivity: Theory, Practice and Training*, páginas 258–265, London, UK, UK, 1995. Chapman & Hall, Ltd.

APÊNDICE A

TOTAL DE MUTANTES GERADOS POR ARQUÉTIPO

Os resultados apresentados nas tabelas A.1, A.2, A.3 e A.4, foram obtidos convertendo-se para OWL apenas a seção DEFINITION de cada arquétipo, sendo necessario ainda converter as outras seções representadas pelos mesmos, assim como, o Modelo de Referência.

A Tabela A.1 apresenta os resultados referentes aos arquétipos *"A health oriented check list"* e *"Informed Consent Request"*.

Tabela A.1: Mutantes gerados para os arquétipos check_list e request

Operador	**Arquétipos**									
	check_list					**request**				
	T[1]	**M**[2]	**E**[3]	**D**[4]	**EM**[5]	**T**	**M**	**E**	**D**	**EM**
CUC	07	03	04	0	1	13	13	0	0	1
CDD	03	03	0	0	1	02	02	0	0	1
CDDO	13	13	0	0	1	66	66	0	0	1
CEU	10	09	0	01	0.9	18	17	0	1	0.944
CED	14	14	0	0	1	66	66	0	0	1
CECO	23	23	0	0	1	127	127	0	0	1
CEUA	08	04	0	04	0.5	0	0	0	0	0
CEUO	06	03	03	0	1	24	12	12	0	1
PDD	03	03	0	0	1	10	01	0	09	0.1
PDUP	05	04	0	01	0.8	03	03	0	0	1
PRD	08	0	08	0	1	12	0	12	0	1
PRUP	02	0	02	0	1	01	0	01	0	1
PDU	06	05	0	01	0.833	04	04	0	0	1
PRU	09	0	0	09	0	14	0	0	14	0
PID	15	15	0	0	1	06	06	0	0	1
ACOTA	04	02	0	02	0.5	17	10	0	07	0.588
ACATO	04	04	0	0	1	0	0	0	0	0
ACSTA	18	14	0	4	0.777	06	06	0	0	1
AEDN	08	08	0	0	1	16	16	0	0	1
TOTAL	**166**	**127**	**17**	**22**	**0,852**	**405**	**349**	**25**	**31**	**0,918**

[1]T - Total de mutantes Gerados.
[2]M - Total de mutantes mortos.
[3]E - Total de mutantes equivalentes.
[4]D - Total de defeitos revelados.
[5]EM - Escore de mutação obtido.

A Tabela A.2 apresenta os resultados referentes aos arquétipos *"Follow up action"* e *"Autopsy examination"*.

Tabela A.2: Mutantes gerados para os arquétipos follow_up e autopsy

Operador	**Arquétipos**									
	follow_up					**autopsy**				
	T	**M**	**E**	**D**	**EM**	**T**	**M**	**E**	**D**	**EM**
CUC	08	08	0	0	1	10	06	04	0	1
CDD	02	02	0	0	1	03	03	0	0	1
CDDO	31	31	0	0	1	27	27	0	0	1
CEU	11	11	0	0	1	16	14	0	2	0.875
CED	31	31	0	0	1	28	28	0	0	1
CECO	56	56	0	0	1	48	48	0	0	1
CEUA	0	0	0	0	0	0	0	0	0	0
CEUO	14	08	06	0	1	08	04	04	0	1
PDD	05	02	0	03	0.4	04	04	0	0	1
PDUP	04	04	0	0	1	08	07	0	01	0.875
PRD	07	0	07	0	1	09	0	09	0	1
PRUP	02	0	02	0	1	03	0	03	0	1
PDU	05	05	0	0	1	08	07	0	01	0.875
PRU	10	0	0	10	0	12	0	0	12	0
PID	10	10	0	0	1	28	28	0	0	1
ACOTA	07	05	0	02	0.714	06	03	0	03	0.5
ACATO	0	0	0	0	0	0	0	0	0	0
ACSTA	06	06	0	0	1	14	14	0	0	1
AEDN	07	07	0	0	1	06	06	0	0	1
TOTAL	**216**	**186**	**15**	**15**	**0,925**	**238**	**199**	**20**	**19**	**0,912**

A Tabela A.3 apresenta os resultados referente aos arquétipos *"Alert"* e *"Fetal Heart Rate"*.

Tabela A.3: Mutantes gerados para o arquétipo alert e fetal_heart

Operador	**Arquétipo**									
	alert					**fetal_heart**				
	T	**M**	**E**	**D**	**EM**	**T**	**M**	**E**	**D**	**EM**
CUC	12	05	07	0	1	10	06	04	0	1
CDD	01	01	0	0	1	52	35	0	17	0.673
CDDO	15	12	0	03	0.8	51	51	0	0	1
CEU	13	11	0	02	0.846	15	15	0	0	1
CED	15	15	0	0	1	51	51	0	0	1
CECO	72	72	0	0	1	86	86	0	0	1
CEUA	0	0	0	0	0	0	0	0	0	0
CEUO	30	30	0	0	1	12	08	04	0	1
PDD	07	07	0	0	1	18	08	0	10	0.444
PDUP	02	02	0	0	1	04	04	0	0	1
PRD	16	0	16	0	1	09	0	09	0	1
PRUP	01	0	01	0	1	03	0	03	0	1
PDU	03	03	0	0	1	07	06	0	01	0.857
PRU	10	0	0	10	0	13	0	0	13	0
PID	03	03	0	0	1	21	21	0	0	1
ACOTA	16	16	0	0	1	07	05	0	02	0.714
ACATO	0	0	0	0	0	0	0	0	0	0
ACSTA	19	19	0	0	1	16	16	0	0	1
AEDN	11	11	0	0	1	08	08	0	0	1
TOTAL	**246**	**207**	**24**	**15**	**0,932**	**383**	**320**	**20**	**43**	**0,881**

A Tabela A.4 apresenta os resultados referente ao arquétipo *"Apgar score"*

Tabela A.4: Mutantes gerados para o arquétipo apgar

Operador	**Arquétipo**				
	apgar				
	T	**M**	**E**	**D**	**EM**
CUC	14	09	05	0	1
CDD	03	03	0	0	1
CDDO	46	46	0	0	1
CEU	19	13	0	06	0.684
CED	46	46	0	0	1
CECO	87	87	0	0	1
CEUA	07	02	0	05	0.285
CEUO	10	06	04	0	1
PDD	11	09	0	02	0.818
PDUP	0	0	0	0	0
PRD	07	0	07	0	1
PRUP	02	0	02	0	1
PDU	07	05	0	02	0.714
PRU	18	01	0	17	0.055
PID	21	21	0	0	1
ACOTA	08	08	0	0	1
ACATO	05	01	0	04	0.2
ACSTA	12	07	0	05	0.583
AEDN	23	22	0	01	0.965
TOTAL	**346**	**286**	**18**	**42**	**0,872**

APÊNDICE B

SEÇÃO DEFINITION DOS ARQUÉTIPOS SELECIONADOS

A Figura B.1 apresenta a codificação ADL da seção DEFINITION do arquétipo ADL openEHR “A health oriented check list”.

```
definition
        EVALUATION[at0000] matches {  -- A health oriented check list
                data matches {
                        ITEM_TREE[at0001] matches {    -- Tree
                                items cardinality matches {1..*; unordered} matches {
                                        CLUSTER[at0004] occurrences matches {1..*} matches {  -- Question group
                                                items cardinality matches {1..*; unordered} matches {
                                                        CLUSTER[at0002] occurrences matches {1..*} matches {  -- Question
                                                                items cardinality matches {1..2; unordered} matches {
                                                                        ELEMENT[at0003] matches {*}
                                                                        ELEMENT[at0005] occurrences matches {0..1} matches {  -- A comment on the answer
                                                                                value matches {
                                                                                        DV_TEXT matches {*}
                                                                                }
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                        ELEMENT[at0006] occurrences matches {0..1} matches {  -- Summary
                                                value matches {
                                                        DV_TEXT matches {*}
                                                }
                                        }
                                }
                        }
                }
        }
```

Figura B.1: Codificação ADL do arquétipo “A health oriented check list”

A Figura B.2 apresenta a codificação ADL da seção DEFINITION do arquétipo ADL openEHR "Informed Consent Request".

```
definition
        INSTRUCTION[at0000] matches {          -- Healthcare service request
                activities cardinality matches {0..*; unordered} matches {
                        ACTIVITY[at0001] occurrences matches {1..*} matches { -- Request
                                description matches {
                                        ITEM_TREE[at0009] matches {   -- Tree
                                                items cardinality matches {1..*; unordered} matches {
                                                        ELEMENT[at0121] matches {      -- Service requested
                                                                value matches {
                                                                        DV_TEXT matches {*}
                                                                }
                                                        }
                                                        ELEMENT[at0135] occurrences matches {0..1} matches {  -- Description of service
                                                                value matches {
                                                                        DV_TEXT matches {*}
                                                                }
                                                        }
                                                        allow_archetype CLUSTER[at0132] occurrences matches {0..*} matches {  -- Specific details
                                                                include
                                                                        archetype_id/value matches {/.*/}
                                                        }
                                                        ELEMENT[at0062] occurrences matches {0..1} matches {  -- Reason for request
                                                                value matches {
                                                                        DV_TEXT matches {*}
                                                                }
                                                        }
                                                        ELEMENT[at0064] occurrences matches {0..1} matches {  -- Reason description
                                                                value matches {
                                                                        DV_TEXT matches {*}
                                                                }
                                                        }
                                                        ELEMENT[at0065] occurrences matches {0..1} matches {  -- Intent
                                                                value matches {
                                                                        DV_TEXT matches {*}
                                                                }
                                                        }
                                                        ELEMENT[at0068] occurrences matches {0..1} matches {  -- Urgency
                                                                value matches {
                                                                        DV_CODED_TEXT matches {
                                                                                defining_code matches {
                                                                                        [local::
                                                                                        at0136, -- Emergency
                                                                                        at0137, -- Urgent
                                                                                        at0138] -- Routine
                                                                                }
                                                                        }
                                                                }
                                                        }
                                                        ELEMENT[at0040] occurrences matches {0..1} matches {  -- Date &/or time service required
                                                                value matches {
                                                                        DV_DATE_TIME matches {*}
                                                                }
                                                        }
                                                        ELEMENT[at0144] occurrences matches {0..1} matches {  -- Latest date service required
                                                                value matches {
                                                                        DV_DATE_TIME matches {*}
                                                                }
                                                        }
                                                        ELEMENT[at0076] occurrences matches {0..1} matches {  -- Supplementary information to follow
                                                                value matches {
                                                                        DV_BOOLEAN matches {
                                                                                value matches {True}
                                                                        }
                                                                }
                                                        }
                                                        ELEMENT[at0078] occurrences matches {0..1} matches {  -- Supplementary information expected
                                                                value matches {
                                                                        DV_TEXT matches {*}
                                                                }
                                                        }
                                                        allow_archetype CLUSTER[at0116] occurrences matches {0..*} matches {  -- Patient requirements
                                                                include
                                                                        archetype_id/value matches {/.*/}
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
```

Figura B.2: Codificação ADL do arquétipo "Informed Consent Request"

A Figura B.3 apresenta a codificação ADL da seção DEFINITION do arquétipo ADL openEHR "Follow up action".

```
definition
        ITEM_TREE[at0000] occurrences matches {0..*} matches {          -- Follow up
                items cardinality matches {1..*; unordered} matches {
                        ELEMENT[at0002] occurrences matches {0..1} matches {  -- Service
                                value matches {
                                        DV_TEXT matches {*}
                                }
                        }
                        ELEMENT[at0004] occurrences matches {0..1} matches {  -- Details
                                value matches {
                                        DV_TEXT matches {*}
                                }
                        }
                        ELEMENT[at0003] occurrences matches {0..1} matches {  -- Appointment date and time
                                value matches {
                                        DV_DATE_TIME matches {
                                                value matches {yyyy-mm-ddTHH:MM:SS}
                                        }
                                }
                        }
                        CLUSTER[at0005] occurrences matches {0..1} matches {  -- Due date
                                items cardinality matches {1..*; unordered} matches {
                                        ELEMENT[at0006] occurrences matches {0..1} matches {  -- Due by absolute date
                                                value matches {
                                                        DV_DATE_TIME matches {*}
                                                }
                                        }
                                        ELEMENT[at0007] occurrences matches {0..1} matches {  -- Due within time interval
                                                value matches {
                                                        DV_INTERVAL<DV_DATE_TIME> matches {
                                                                upper matches {
                                                                        DV_DATE_TIME matches {*}
                                                                }
                                                                lower matches {
                                                                        DV_DATE_TIME matches {*}
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
```

Figura B.3: Codificação ADL do arquétipo "Follow up action"

A Figura B.4 apresenta a codificação ADL da seção DEFINITION do arquétipo ADL openEHR “Autopsy examination”.

```
definition
    OBSERVATION[at0000] matches {          -- Autopsy examination
        data matches {
            HISTORY[at0001] matches {      -- history
                events cardinality matches {1..*; unordered} matches {
                    EVENT[at0002] occurrences matches {0..*} matches {     -- Any event
                        data matches {
                            ITEM_TREE[at0003] matches {   -- Tree
                                items cardinality matches {0..1; ordered} matches {
                                    ELEMENT[at0004] occurrences matches {0..1} matches {  -- Clinical
                                        value matches {
                                            DV_TEXT matches {*}
                                        }
                                    }
                                    ELEMENT[at0005] occurrences matches {0..1} matches {  -- External examination
                                        value matches {
                                            DV_TEXT matches {*}
                                        }
                                    }
                                    CLUSTER[at0006] occurrences matches {0..1} matches {  -- Internal examination
                                        items cardinality matches {1..*; ordered} matches {
                                            ELEMENT[at0007] occurrences matches {0..*} matches {  -- System
                                                name matches {
                                                    DV_CODED_TEXT matches {
                                                        defining_code matches {
                                                            [local::
                                                            at0008, -- Cardiovascular system
                                                            at0009, -- Respiratory system
                                                            at0010, -- Gastro-intestinal system
                                                            at0011, -- Reticulo-Endothelial system
                                                            at0012, -- Genito-urinary system
                                                            at0013, -- Endocrine System
                                                            at0014, -- Central nervous system
                                                            at0015] -- Musculoskeletal system
                                                        }
                                                    }
                                                }
                                                value matches {
                                                    DV_TEXT matches {*}
                                                }
                                            }
                                        }
                                    }
                                    CLUSTER[at0016] occurrences matches {0..1} matches {  -- Microscopic examination
                                        items cardinality matches {1..*; ordered} matches {
                                            ELEMENT[at0017] occurrences matches {0..*} matches {  -- Organ
                                                name matches {
                                                    DV_CODED_TEXT matches {
                                                        defining_code matches {
                                                            [local::
                                                            at0018, -- Adrenals
                                                            at0019, -- Brain
                                                            at0020, -- Heart
                                                            at0021, -- Kidneys
                                                            at0022, -- Liver
                                                            at0023, -- Lungs
                                                            at0024, -- Pancreas
                                                            at0025, -- Pituitary
                                                            at0026, -- Small bowel
                                                            at0027, -- Spleen
                                                            at0028, -- Thyroid
                                                            at0029] -- Vertebra
                                                        }
                                                    }
                                                }
                                                value matches {
                                                    DV_TEXT matches {*}
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
```

Figura B.4: Codificação ADL do arquétipo “Autopsy examination”

A Figura B.5 apresenta a codificação ADL da seção DEFINITION do arquétipo ADL openEHR "Alert".

```
definition
        EVALUATION[at0000] matches {  -- Alert
                data matches {
                        ITEM_LIST[at0001] matches {    -- List
                                items cardinality matches {1..*; unordered} matches {
                                        ELEMENT[at0015] occurrences matches {0..1} matches {  -- Alert
                                                value matches {
                                                        DV_TEXT matches {*}
                                                }
                                        }
                                        ELEMENT[at0002] occurrences matches {0..1} matches {  -- Category
                                                value matches {
                                                        DV_TEXT matches {*}
                                                }
                                        }
                                        ELEMENT[at0003] matches {      -- Description
                                                value matches {
                                                        DV_TEXT matches {*}
                                                }
                                        }
                                        ELEMENT[at0009] occurrences matches {0..1} matches {  -- Status
                                                value matches {
                                                        DV_CODED_TEXT matches {
                                                                defining_code matches {
                                                                        [local::
                                                                        at0011, -- Active
                                                                        at0012, -- Inactive
                                                                        at0013] -- Resolved
                                                                }
                                                        }
                                                }
                                        }
                                        ELEMENT[at0004] occurrences matches {0..1} matches {  -- Start Date
                                                value matches {
                                                        DV_DATE_TIME matches {*}
                                                }
                                        }
                                        ELEMENT[at0010] occurrences matches {0..1} matches {  -- Review Date
                                                value matches {
                                                        DV_DATE_TIME matches {*}
                                                }
                                        }
                                        ELEMENT[at0014] occurrences matches {0..1} matches {  -- End Date
                                                value matches {
                                                        DV_DATE_TIME matches {*}
                                                }
                                        }
                                }
                        }
                }
        }
```

Figura B.5: Codificação ADL do arquétipo "Alert"

A Figura B.6 apresenta a codificação ADL da seção DEFINITION do arquétipo ADL openEHR “Fetal Heart Rate”.

```
definition
        OBSERVATION[at0000] matches {         -- Fetal Heart Rate
                subject matches {
                        PARTY_RELATED matches {
                                relationship matches {
                                        DV_CODED_TEXT matches {
                                                defining_code matches {[openehr::3]}
                                        }
                                }
                        }
                }
                data matches {
                        HISTORY[at0001] matches {      -- Event Series
                                events cardinality matches {1..*; unordered} matches {
                                        EVENT[at0002] occurrences matches {0..1} matches {     -- Any event
                                                data matches {
                                                        ITEM_TREE[at0003] matches {   -- Tree
                                                                items cardinality matches {0..*; unordered} matches {
                                                                        ELEMENT[at0004] occurrences matches {0..1} matches {  -- Heart beat present?
                                                                                value matches {
                                                                                        DV_BOOLEAN matches {
                                                                                                value matches {True, False}
                                                                                        }
                                                                                }
                                                                        }
                                                                        ELEMENT[at0005] occurrences matches {0..1} matches {  -- Heart Rate
                                                                                value matches {
                                                                                        C_DV_QUANTITY <
                                                                                                property = <[openehr::382]>
                                                                                                list = <
                                                                                                        ["1"] = <
                                                                                                                units = <"/min">
                                                                                                                magnitude = <|>=0.0|>
                                                                                                                precision = <|0|>
                                                                                                        >
                                                                                                >
                                                                                        >
                                                                                }
                                                                        }
                                                                        ELEMENT[at0009] occurrences matches {0..1} matches {  -- Comment
                                                                                value matches {
                                                                                        DV_TEXT matches {*}
                                                                                }
                                                                        }
                                                                        ELEMENT[at0010] occurrences matches {0..1} matches {  -- Conclusion
                                                                                value matches {
                                                                                        DV_TEXT matches {*}
                                                                                }
                                                                        }
                                                                }
                                                        }
                                                }
                                                state matches {
                                                        ITEM_TREE[at0011] matches {   -- Tree
                                                                items cardinality matches {0..*; unordered} matches {
                                                                        ELEMENT[at0012] occurrences matches {0..1} matches {  -- Position of Mother
                                                                                value matches {
                                                                                        DV_CODED_TEXT matches {
                                                                                                defining_code matches {
                                                                                                        [local::
                                                                                                        at0013, -- Standing
                                                                                                        at0014, -- Sitting
                                                                                                        at0015, -- Reclining
                                                                                                        at0016, -- Lying
                                                                                                        at0017] -- Lying with tilt to left
                                                                                                }
                                                                                        }
                                                                                }
                                                                        }
                                                                        ELEMENT[at0019] occurrences matches {0..1} matches {  -- Maternal Confounding Factors
                                                                                value matches {
                                                                                        DV_TEXT matches {*}
                                                                                }
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
```

Figura B.6: Codificação ADL do arquétipo “Fetal Heart Rate”

A Figura B.7 apresenta a codificação ADL da seção DEFINITION do arquétipo ADL openEHR “Apgar score”.

```
definition
    OBSERVATION[at0000] matches {        -- Apgar score
        data matches {
            HISTORY[at0002] matches {       -- history
                events cardinality matches {1..*; unordered} matches {
                    POINT_EVENT[at0003] occurrences matches {0..1} matches {      -- 1 minute
                        offset matches {
                            DV_DURATION matches {
                                value matches {|PT1M|}
                            }
                        }
                        data matches {
                            ITEM_LIST[at0001] matches {    -- structure
                                items cardinality matches {1..6; ordered} matches {
                                    ELEMENT[at0009] occurrences matches {0..1} matches {  -- Respiratory effort
                                        value matches {
                                            0|[local::at0010],        -- Absent
                                            1|[local::at0011],        -- Weak or irregular
                                            2|[local::at0012]         -- Normal
                                        }
                                    }
                                    ELEMENT[at0005] occurrences matches {0..1} matches {  -- Heart Rate
                                        value matches {
                                            0|[local::at0006],        -- Absent
                                            1|[local::at0007],        -- <100 beats per minute
                                            2|[local::at0008]         -- ≥100 beats per minute
                                        }
                                    }
                                    ELEMENT[at0013] occurrences matches {0..1} matches {  -- Muscle tone
                                        value matches {
                                            0|[local::at0014],        -- Limp or flaccid
                                            1|[local::at0015],        -- Reduced tone
                                            2|[local::at0016]         -- Normal tone
                                        }
                                    }
                                    ELEMENT[at0017] occurrences matches {0..1} matches {  -- Reflex irritability
                                        value matches {
                                            0|[local::at0018],        -- No response
                                            1|[local::at0019],        -- Reduced response
                                            2|[local::at0020]         -- Normal response
                                        }
                                    }
                                    ELEMENT[at0021] occurrences matches {0..1} matches {  -- Skin colour
                                        value matches {
                                            0|[local::at0022],        -- Completely blue
                                            1|[local::at0023],        -- Body pink; extremities blue
                                            2|[local::at0024]         -- Completely pink
                                        }
                                    }
                                    ELEMENT[at0025] occurrences matches {0..1} matches {  -- Total
                                        value matches {
                                            DV_COUNT matches {
                                                magnitude matches {|0..10|}
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                    POINT_EVENT[at0026] occurrences matches {0..1} matches {      -- 2 minute
                        offset matches {
                            DV_DURATION matches {
                                value matches {|PT2M|}
                            }
                        }
                        data matches {
                            use_node ITEM_LIST /data[at0002]/events[at0003]/data[at0001]  -- /data[history]/events[1 minute]/data[structure]
                        }
                    }
                    POINT_EVENT[at0027] occurrences matches {0..1} matches {      -- 3 minute
                        offset matches {
                            DV_DURATION matches {
                                value matches {|PT3M|}
                            }
                        }
                        data matches {
                            use_node ITEM_LIST /data[at0002]/events[at0003]/data[at0001]  -- /data[history]/events[1 minute]/data[structure]
                        }
                    }
                    POINT_EVENT[at0028] occurrences matches {0..1} matches {      -- 5 minute
                        offset matches {
                            DV_DURATION matches {
                                value matches {|PT5M|}
                            }
                        }
                        data matches {
                            use_node ITEM_LIST /data[at0002]/events[at0003]/data[at0001]  -- /data[history]/events[1 minute]/data[structure]
                        }
                    }
                    POINT_EVENT[at0031] occurrences matches {0..1} matches {      -- 10 minute
                        offset matches {
                            DV_DURATION matches {
                                value matches {|PT10M|}
                            }
                        }
                        data matches {
                            use_node ITEM_LIST /data[at0002]/events[at0003]/data[at0001]  -- /data[history]/events[1 minute]/data[structure]
                        }
                    }
                    EVENT[at0037] occurrences matches {0..*} matches {     -- Any event
                        data matches {
                            use_node ITEM_LIST /data[at0002]/events[at0003]/data[at0001]  -- /data[history]/events[1 minute]/data[structure]
                        }
                    }
                }
            }
        }
        protocol matches {
            ITEM_LIST[at0029] matches {    -- List
                items cardinality matches {0..*; unordered} matches {
                    ELEMENT[at0030] occurrences matches {0..1} matches {  -- Notes on measurement
                        value matches {
                            DV_TEXT matches {*}
                        }
                    }
                }
            }
        }
    }
```

Figura B.7: Codificação ADL do arquétipo “Apgar score”

APÊNDICE C

ONTOLOGIAS OWL DOS ARQUÉTIPOS SELECIONADOS

A Figura C.1 representa a estrutura de classes, propriedades de objetos e propriedades de dados da ontologia OWL obtida através da conversão do arquétipo ADL openEHR "A health oriented check list".

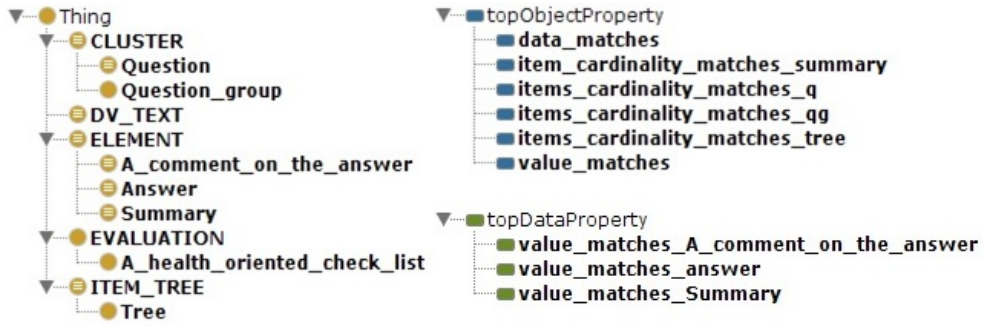

Figura C.1: Ontologia OWL referente ao arquétipo "A health oriented check list"

A Figura C.2 representa a estrutura de classes, propriedades de objetos e propriedades de dados da ontologia OWL obtida através da conversão do arquétipo ADL openEHR "Informed Consent Request".

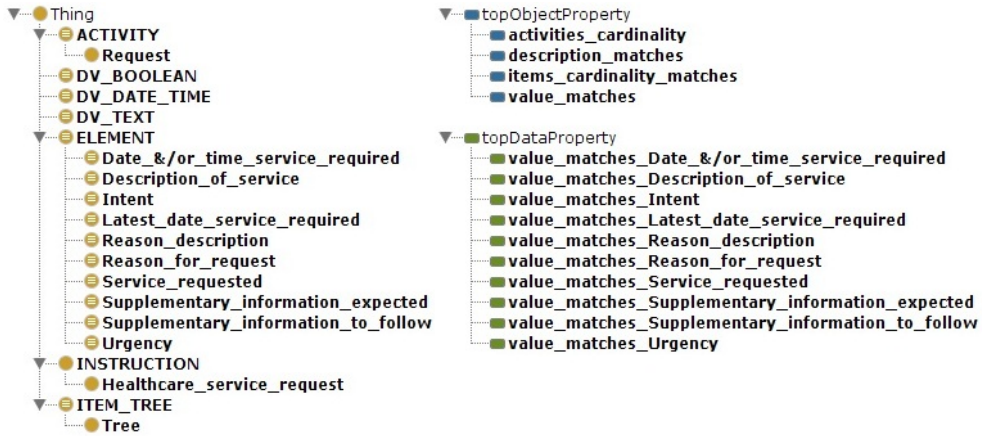

Figura C.2: Ontologia OWL referente ao arquétipo "Informed Consent Request"

A Figura C.3 representa a estrutura de classes, propriedades de objetos e propriedades de dados da ontologia OWL obtida através da conversão do arquétipo ADL openEHR "Follow up action".

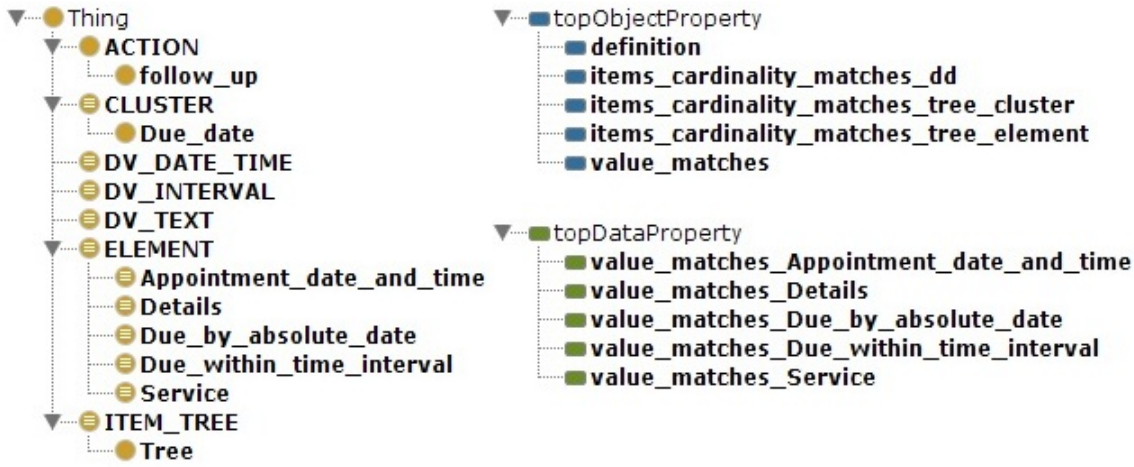

Figura C.3: Ontologia OWL referente ao arquétipo "Follow up action"

A Figura C.4 representa a estrutura de classes, propriedades de objetos e propriedades de dados da ontologia OWL obtida através da conversão do arquétipo ADL openEHR "Autopsy examination".

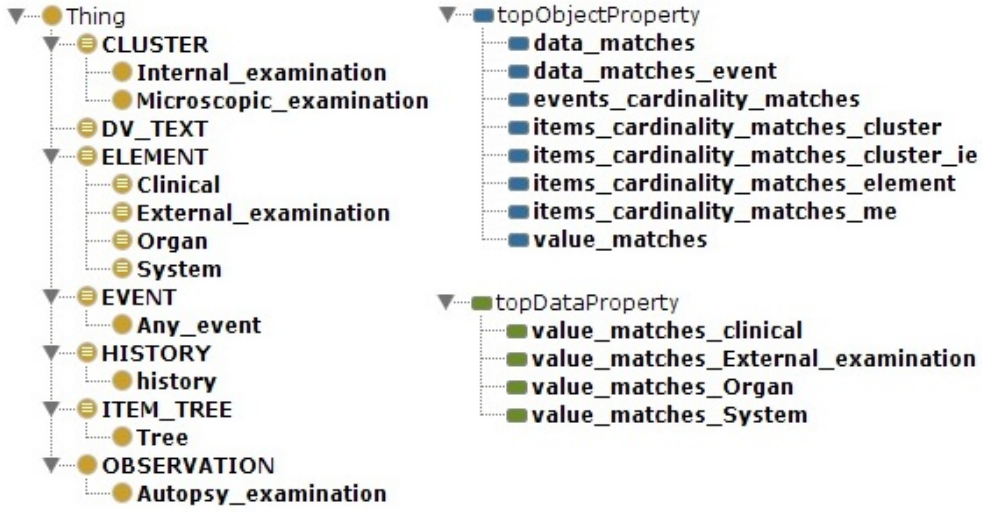

Figura C.4: Ontologia OWL referente ao arquétipo "Autopsy examination"

A Figura C.5 representa a estrutura de classes, propriedades de objetos e propriedades de dados da ontologia OWL obtida através da conversão do arquétipo ADL openEHR "Alert".

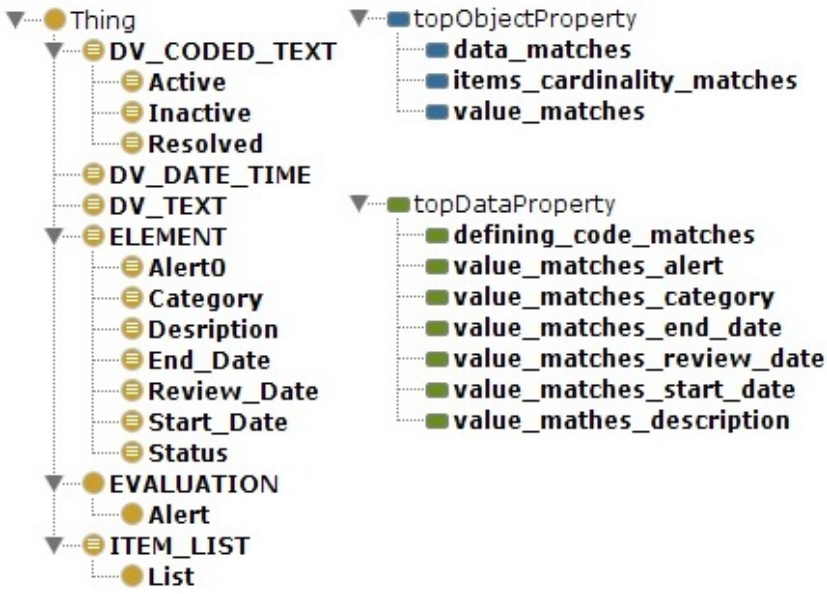

Figura C.5: Ontologia OWL referente ao arquétipo "Alert"

A Figura C.6 representa a estrutura de classes, propriedades de objetos e propriedades de dados da ontologia OWL obtida através da conversão do arquétipo ADL openEHR "Fetal Heart Rate".

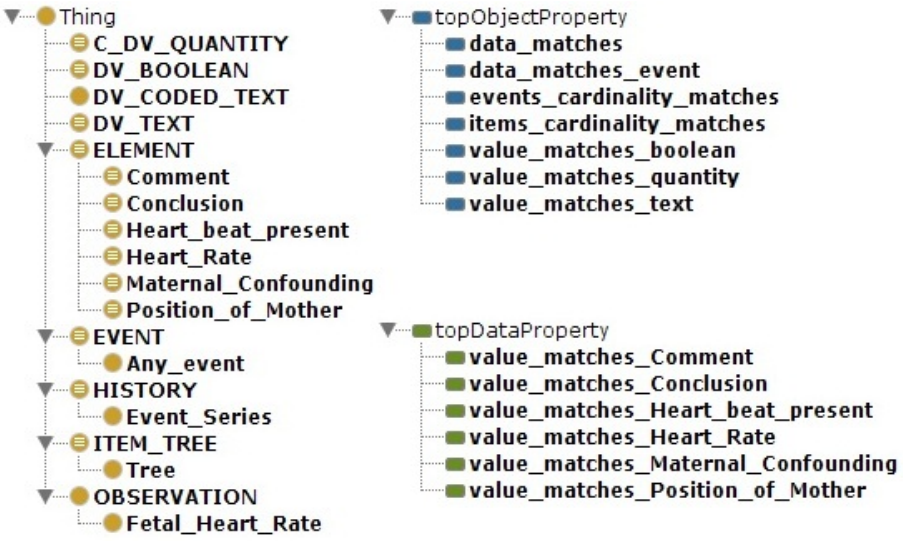

Figura C.6: Ontologia OWL referente ao arquétipo "Fetal Heart Rate"

A Figura C.7 representa a estrutura de classes, propriedades de objetos e propriedades de dados da ontologia OWL obtida através da conversão do arquétipo ADL openEHR "Apgar score".

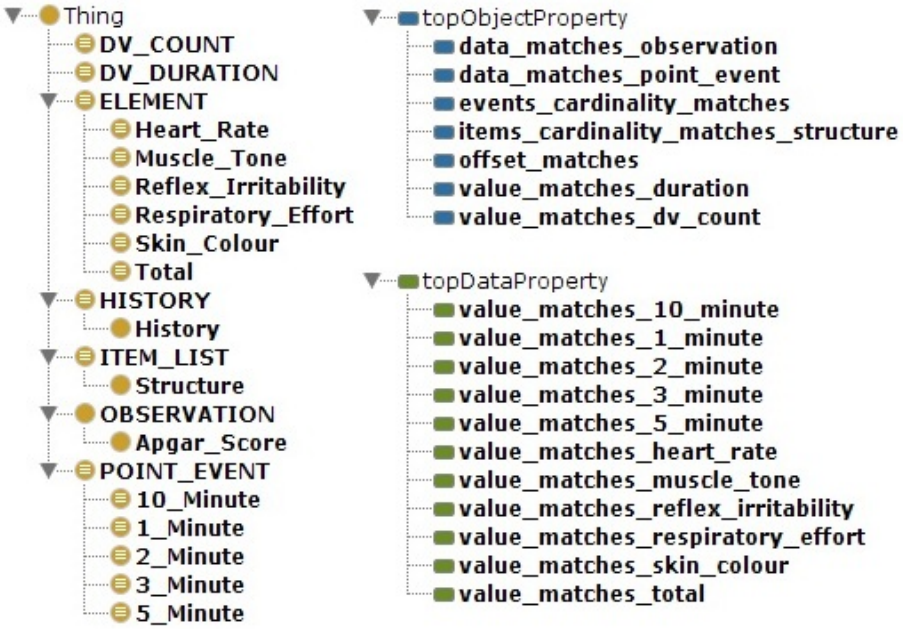

Figura C.7: Ontologia OWL referente ao arquétipo "Apgar score"