Pós-Graduação em Ciência da Computação

DIOGO ESPINHARA OLIVEIRA

COREACQ:

um *framework* computacional para validar questões de competência por raciocínio automático sobre a ontologia SUMO

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife

2019

DIOGO ESPINHARA OLIVEIRA

COREACQ:

um *framework* computacional para validar questões de competência por raciocínio automático sobre a ontologia SUMO

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial

Orientador: Prof. Dr. Frederico Luiz Gonçalves de Freitas

Coorientador: Prof. Dr. Ryan Ribeiro de Azevedo

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

O48c Oliveira, Diogo Espinhara
COREACQ: um *framework* computacional para validar questões de competência por raciocínio automático sobre a ontologia SUMO / Diogo Espinhara Oliveira. – 2019.
102 f.: il., fig., tab.

Orientador: Frederico Luiz Gonçalves de Freitas.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências e apêndice.

1. Inteligência artificial. 2. Ontologia. 3. Raciocínio automático. I. Freitas, Frederico Luiz Gonçalves de (orientador). II. Título.

006.3 CDD (23. ed.) UFPE- MEI 2019-040

DIOGO ESPINHARA OLIVEIRA

COREACQ:

um *framework* computacional para validar questões de competência por raciocínio automático sobre a ontologia SUMO

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 27/02/2019

BANCA EXAMINADORA

Prof. Frederico Luiz Gonçalves de Freitas (Orientador)
Prof. UFPE / Centro de Informática

Profa. Anjolina Grisi de Oliveira (Examinador Interno)
UFPE / Centro de Informática

Prof. José Maria Parente de Oliveira (Examinador Externo)
ITA / Departamento de Ciência da Computação

À minha família e esposa que me apoiaram
perante as dificuldades durante este percurso,
Dedico

AGRADECIMENTOS

Aos meus pais, meu irmão e minha esposa, agradeço com todo o amor possível e impossível. Doriano, Edilene, Diego, e Emillin que nunca mediram esforços para dar suporte e apoio a mim; conseguimos realizar esse sonho! Chegamos ao fim de uma intensa jornada acadêmica, saibam que os sacrifícios não foram em vão. Doriano e Edilene, meus pais, me tornei o que vocês um dia sonharam para mim, sou um Mestre, o seu Mestre. Orgulho, Doriano e Edilene, por ter conseguido retribuir o esforço de vocês com uma conquista tão grandiosa para a nossa família. Emillin, minha esposa, ganhamos essa batalha, que não seria possível completar sem o seu apoio e amor incondicional, tudo que é meu é seu, dito isso, conseguimos, nós somos Mestres.

Ryan Ribeiro, mentor e amigo excepcional, sem igual, uma pessoa fora do comum. Como orientador sempre me apoiou e forneceu todos os recursos necessários para a realização de meu sonho. Ryan, você é um dos professores notáveis que me inspiraram a aprofundar meus conhecimentos dentro da área acadêmica. Obrigado por ter sido tão próximo, por ter sido líder e me cobrar quando deveria. Você um dos grandes responsáveis por meu crescimento profissional e, também, pessoal. Agradeço por proporcionar minha evolução acadêmica e pela oportunidade de ser orientado por você.

Fred Freitas, um professor e pesquisador brilhante, fora do comum, um ser humano incomparável. Como professor é excepcional, transmitindo seus valorosos conhecimentos que fomentam a academia e a indústria privada, inspirando seus alunos a buscarem novos saberes. Como pesquisador é imprescindível, em sua área de atuação é um nome reconhecido por contribuir com novas pesquisas e descobertas que fomentam o desenvolvimento acadêmico e da indústria. Quando me apresentou o problema a ser tratado por esta dissertação, fiquei animado e admirado com o tema proposto. O resultado foi este, uma dissertação única. Fred, muito obrigado por ter me permitido participar de seu grupo de pesquisa e por ser hoje um dos responsáveis por meus sucessos acadêmicos e profissionais. Obrigado por ter sido meu orientador e pela oportunidade de desenvolver esse projeto contigo.

Aos meus três eternos amigos, Marcos Aurério, Ricardo e Edvanio, com quem convivi em minha trajetória acadêmica desde a UFRPE em Garanhuns-PE. Durante anos tive o prazer de conviver com vocês nas alegrias e, também, nos desânimos da vida. Obrigado a vocês pelo apoio e suporte em minha trajetória de vida.

Ao Centro de Informática da Universidade Federal de Pernambuco - CIN-UFPE - e seus colaboradores, aos Diretores do CIN e Coordenadores da Pós-Graduação. Um agradecimento especial a todos os professores do curso de mestrado do CIN-UFPE, que contribuíram direto e indiretamente para meu crescimento acadêmico e profissional.

Ao CNPq pela bolsa de mestrado, um recurso financeiro indispensável para a conclusão

deste curso de Mestrado. Consegui cumprir meu papel e fazer jus à bolsa que me foi concedida. Obrigado CNPq pelo suporte financeiro.

RESUMO

Abordagens baseadas em Questões de Competência (CQ, *Competency Question*), que permitem especificar os requisitos de uma base de conhecimento na forma de consultas, usadas para a avaliação de ontologias, são bastante utilizadas em ferramentas encontradas na área da Engenharia de Ontologias. Um engenheiro de ontologias deve verificar a sua ontologia de acordo com a especificação de seu projeto, para isso, pode definir um conjunto de CQs que deve ser inferido - isto é, confirmado por raciocínio automático - pela ontologia e, no caso de informações inconsistentes ou incompletas, precisa corrigir os problemas encontrados. Atualmente existem ferramentas capazes de apoiar o processo de avaliação de ontologias através de recursos para facilitar e automatizar a verificação (ou validação) de CQs, entretanto, ainda necessitam de muita intervenção humana para solucionar as falhas no desenvolvimento das ontologias. Este quadro situacional leva nossa exploração a meios de possibilitar que ferramentas possam evoluir uma ontologia de forma automática, fazendo uso de fontes de informações confiáveis e gratuitas como, por exemplo, a Ontologia de Topo SUMO. Desenvolvemos o *CoreACQ*, um *framework* computacional, projetado para validar CQs por raciocínio automático sobre a SUMO. Nossa solução consiste em uma solução viável para o problema de ontologias de domínio incompletas - as quais falham no processo de avaliação por não possuírem os conhecimentos exigidos como requisitos. Os resultados alcançados demonstraram que *CoreACQ* consiste em uma solução eficiente para: (1) Validação de CQs; para isso, manipula consultas em FOL (*First Order Logic*) e realiza inferências sobre a SUMO com o objetivo de respondê-las utilizando um sistema ATP (*Automated Theorem Prover*) e para (2) Raciocínio Automático; as funcionalidades implementadas permitem que novos fatos sejam deduzidos a partir de uma ontologia em FOL, bem como otimização do tempo do processo de raciocínio por representação e busca de axiomas em grafos e um mecanismo de *cache*. Concluímos também que nosso *framework* é uma solução computacional que pode ser utilizado por outras ferramentas de desenvolvimento de ontologias, como por exemplo, o *protégé*.

Palavras-chave: Ontologia de Topo. SUMO. Questão de Competência. Raciocínio Automático. Avaliação de Ontologia. Representação de Conhecimento.

ABSTRACT

Approaches based on Competency Questions (CQ), that allow to specify the requirements of an knowledge base in the form of queries, designed for ontology evaluation, are widely used in tools found in the area of ontology engineering. An ontology engineer must to verify your ontology according to the specification of your project, he can to define a set of CQs which must be inferred - confirmed by automated reasoning - by the ontology and, in the case of inconsistent or incomplete informations, he needs to fix the found problems. Nowadays there are tools capable of support the process for ontology evaluation through resources to facilitate and automate the verification (or validation) of CQs, however, they have needed a lot of intervention from humans to fix errors in the development of the ontologies. This situation take us to an exploration by new ways to allow tools that can improve an ontology automatically, using free and reliable sources of information like, for example, the SUMO Upper Ontology. We develop CoreACQ, a computational framework, built to validate CQs by automated reasoning over the SUMO. Our solution can be a feasible solution to the problem of incomplete domain ontologies - those who fail in the evaluation process because don't have the required knowledge. The results obtained show that CoreACQ is an efficient solution to: (1) CQ Validation; it manipulates FOL (First Order Logic) queries and performs inferences over the SUMO with the purpose of answering them through an ATP (Automated Theorem Prover) system and (2) Automated Reasoning; the implemented functions allow deduction of knowledge from a FOL ontology, and optimization of the reasoning process time by axioms representation and search in graphs and a cache mechanism. We also conclude that our framework is an computacional solution that can be used by other ontology development tools, like for example, the protégé.

Keywords: Upper Ontology. SUMO. Competency Question. Automated Reasoning. Ontology Evaluation. Knowledge Representation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ABox	Assertional Box
ACE	Attempto Controlled English
ALC	Attributive Concept Language with Complements
ATP	Automated Theorem Prover
CNL	Controlled Natural Languages
CQ	Competency Question
DL	Description Logics
FOF	First Order Formula
FOL	First Order Logic
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IA	Inteligência Artificial
KB	Knowledge Base
KIF	Knowledge Interchange Format
LN	Linguagem Natural
OWL	Ontology Web Language
OWL 2 DL	OWL 2 Description Logics
OWL 2	Ontology Web Language 2
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SUMO	Suggested Upper Merged Ontology
SUO	Suggested Upper Ontology
SUO WG	Standard Upper Ontology Working Group
TBox	Terminological Box
THF	Typed Higher-Order Form
TPTP	Thousands of Problems for Theorem Provers
W3C	World Wide Web Consortium
Web	World Wide Web
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

A busca na *Web* proporciona um recurso essencial para descoberta de informações por parte dos humanos. A recuperação de informação baseada em palavras-chave é um dos principais métodos usados por engenhos de busca como *Google*[1]. Esse tipo de técnica extrai as palavras da consulta e verifica a presença delas nos documentos para decidir quais serão recuperados, isto é, sem analisar os seus significados. À vista disso, pesquisadores das áreas de Ciência da Computação e Ciência da Informação vêm expondo a necessidade de definir precisamente os conceitos na *Web*.

Uma contribuição relevante das áreas de Ciência da Computação e Ciência da Informação refere-se às ontologias, arquivos de computador que descrevem objetos de um domínio e os relacionamentos entre eles (BERNERS-LEE; HENDLER; LASSILA, 2001). Por meio delas, uma máquina computacional é capaz de entender o sentido dos conceitos e descobrir conhecimentos não explícitos nos documentos. Em razão disso, as ontologias constituem um dos principais componentes da *Semantic Web* (em tradução livre Web Semântica), uma extensão da *Web* convencional que define formalmente o significado de suas informações.

À medida que as ontologias ganhavam mais visibilidade, um grupo diversificado de pessoas percebeu a necessidade de grandes ontologias públicas e gratuitas. Por esse motivo, em 2001, foi fundado o *Standard Upper Ontology Working Group* (SUO WG, em tradução livre Grupo de Trabalho para a Ontologia de Topo Padrão). Ele foi formado por colaboradores das áreas de Engenharia, Filosofia, e Ciência da Informação. O seu objetivo relaciona-se a criação da *Standard Upper Ontology* (SUO, em tradução livre Ontologia de Topo Padrão), uma ontologia da categoria conhecida como *nível superior* ou *topo*. Tal categoria destina-se a definições sobre conceitos de propósito geral, que atuam como base para ontologias de domínios mais específicos (NILES; PEASE, 2001a).

Em seguinda, após alguns meses, ocorreu a publicação da SUMO[2] (*Suggested Upper Merged Ontology*, Ontologia Mesclada de Topo Sugerida) com o propósito de ser um documento inicial para o SUO WG (NILES; PEASE, 2001b). Mesmo após vários anos, a SUMO ainda continua sendo desenvolvida e atualmente possui cerca de 25 mil termos e 80 mil axiomas.

Outro tipo de ontologia são as ontologias de domínio, elas referem-se a definições de conceitos próprios (direito tributário) sobre uma área mais genérica (direito financeiro). Em razão de resolver problemas específicos, tornou-se a categoria com mais atenção entre os pesquisadores da engenharia de ontologias. Entretanto, eles não restringem-se ao nível de domínio, isto é, também abordam padrões e metodologias referentes a outros tipos de

[1] https://www.google.com/
[2] http://www.adampease.org/OP/

ontologias.

No que diz respeito a padrões e metodologias para o desenvolvimento de ontologias, W3C[3] define a OWL (*Ontology Web Language*) como a linguagem padrão para criação e manipulação de ontologias na *Web*. Ela é dividida em sublinguagens de acordo com suas expressividades, sendo uma delas compatível com DL (*Description Logic*, em tradução livre Lógica de Descrição) (MCGUINNESS; HARMELEN, 2004). Pesquisas na área também definem como especificar o ciclo de vida, descrever o escopo e requisitos, criar uma especificação, conduzir a evolução, e outras atividades fundamentais para o desenvolvimento de uma ontologia. Sobre as metodologias, é relevante citar, por exemplo, *Methontology* (FERNÁNDEZ-LÓPEZ; GÓMEZ-PÉREZ; JURISTO, 1997), *On-To-Knowledge* (STAAB et al., 2001), e *Ontology 101* (NOY; MCGUINNESS, 2001).

Apesar das metodologias sobre ontologia possuírem suas diferenças, elas compartilham do mesmo propósito de organizar o processo de desenvolvimento em passos bem definidos. Além disso, existem várias ferramentas com a finalidade de apoiar a engenharia de ontologias, algumas delas são: Protégé (GENNARI et al., 2003), OntoStudio [4], NeOn Toolkit (HAASE et al., 2008), OntoEdit (SURE et al., 2002), e WebODE (ARPÍREZ et al., 2001).

Mesmo com avanços na engenharia de ontologias, ainda surgem situações inadequadas como, por exemplo, conhecimento insuficiente sobre um determinado domínio de conhecimento (MALHEIROS; FREITAS, 2017). Para resolver ou minimizar esses problemas pode-se considerar as *Competency Questions* (CQs, em tradução livre Questões de Competência), um conjunto de questões que uma base de conhecimento deve ser capaz de responder usando seus axiomas. Cada questão pode ser entendida como um requisito na forma de uma consulta em linguagem natural ou formal. Dessa maneira, um engenheiro pode definir requisitos e avaliar se uma ontologia está em concordância com a especificação do seu escopo e domínio.

Para exemplificar, uma CQ do tipo "vaca é um herbívoro?" deveria, a priori, ser validada - isto é, deduzida - no domínio dos herbívoros. Portanto, são importantes para garantir que a ontologia possua o conhecimento necessário para resolver o problema de interesse. Não obstante, a avaliação de requisitos, se feita manualmente, torna-se uma tarefa dispendiosa e propensa a mais erros (FREITAS; MALHEIROS, 2013).

Tendo em consideração o processo de avaliação de uma ontologia, é essencial o emprego de uma máquina de inferência para executar raciocínio automático - isto é, dedução de novos conhecimentos de forma autônoma. Com isso, informações necessárias para validar uma CQ podem ser deduzidas através da própria ontologia, agilizando a sua evolução e minimizando custos de desenvolvimento. Sobre OWL 2 DL, podemos idealizar o emprego

[3] O World Wide Web Consortium (W3C) é uma comunidade internacional onde as organizações membros, uma equipe de tempo integral e o público trabalham em conjunto para desenvolver padrões da Web

[4] http://www.semafora-systems.com/en/products/ontostudio/

de máquinas de inferência como *Pellet* [5] (PARSIA; SIRIN, 2004), *HermiT* [6] (SHEARER; MOTIK; HORROCKS, 2008) e *RACCOON*[7] (FILHO; FREITAS; OTTEN, 2017).

Apesar da utilização de máquinas de inferência, é possível existir uma ontologia sem informações suficientes para a etapa de sua avaliação. Para resolver ou reduzir esse problema, pode-se considerar o uso de ontologias de topo, que descrevem conceitos e axiomas gerais (e.g. "*Humanos são mamíferos*") que são verdadeiros para uma grande variedade de domínios. Algumas delas encontram-se disponíveis por meio da Internet, como: Ontology4 [8]; BFO [9]; e SUMO [10]. Por isso, tornam-se bastante reusáveis por engenheiros de ontologias que desejam agilizar seu trabalho.

Vejamos a ontologia de topo SUMO, sua linguagem para representação do conhecimento é denominada SUO-KIF (*Standard Upper Ontology Knowledge Interchange Format*, em tradução livre Formato para Troca de Conhecimento da Ontologia de Topo Padrão), que foi idealizada pelo SUO WG para servir de formalismo adequado para a SUO (NILES; PEASE, 2001b). No entanto, possui versões desenvolvidas em lógica de primeira ordem (FOL, *First Order Logic*), como, por exemplo, a *Adimen-SUMO*, construída através de reengenharia de aproximadamente 88% da versão original (ÁLVEZ; LUCIO, 2012).

Em relação a raciocínio automático sobre a SUMO, faz-se necessário o uso de um sistema ATP (*Automatic Theorem Prover*, em tradução livre Provador Automático de Teoremas), como *E Prover*[11] (SCHULZ, 2013) e *Vampire Prover*[12] (KOVÁCS; VORONKOV, 2013). Ambos funcionam com fórmulas em FOL para provar afirmações lógicas e matemáticas, entretando sem suporte direto a SUO-KIF - isto é, demanda conversão entre as duas linguagens. Contudo, Álvez, Lucio e Rigau (2015) conduziram experimentos mostrando que a ontologia *Adimen-SUMO v2.4*, formalizada em FOL, desempenhou os melhores resultados se comparada a versão em SUO-KIF.

Neste trabalho, entende-se que as informações derivadas de ontologias de topo, e.g. SUMO, podem proporcionar autonomia para ferramentas evoluírem ontologias de domínio evitando intervenção humana. Assim sendo, na Subseção 1.1, a seguir, é apresentado a motivação para a realização deste trabalho.

1.1 Contexto e Motivação

A importância de se avaliar a capacidade de uma ontologia em representar um domínio vem sendo consolidada por metodologias como *Methontology*, *On-To-Knowledge*, e *Ontology 101*. As questões de competência proporcionam um mecanismo para um engenheiro

[5] https://www.w3.org/2001/sw/wiki/Pellet
[6] http://www.hermit-reasoner.com/
[7] https://github.com/dmfilho/raccoon
[8] http://www.ontology4.us/
[9] http://ifomis.uni-saarland.de/bfo/
[10] http://www.adampease.org/OP/
[11] http://www.eprover.org/
[12] http://www.vprover.org/

definir os requisitos de uma ontologia e conferir os conhecimentos representados em sua base de conhecimento. Sobre as ferramentas, tem-se observado um esforço progressivo para desenvolver procedimentos capazes de validar requisitos por intermédio de CQs. Na atualidade, encontram-se diversas pesquisas com tal propósito, algumas são discutidas por: Freitas e Malheiros (2013); Carmen, Pradel e Hernandez (2012); Bezerra, Santana e Freitas (2013); Freitas e Malheiros (2013); e Ren et al. (2014).

Atualmente, observa-se ainda a existência de ferramentas capazes de identificar informações que faltam ser esclarecidas na avaliação de uma ontologia. Malheiros e Freitas (2014) apresentam um sistema com a capacidade de especificar e propor questões a serem respondidas antes do processo de avaliação continuar. Assim, ao invés de passivo, a engenharia de ontologias dispõe de autonomia para determinar quais conhecimentos não estão em uma ontologia e são essenciais para a etapa de sua avaliação.

Ainda que ferramentas de ontologia disponibilizem a validação e, até, a geração de CQs, há o problema de suspenção temporária do processo de avaliação devido a questões não respondidas pela ontologia. Quando isso ocorre, a responsabilidade de resolver a falta de informação é atribuída integralmente ao engenheiro de ontologias. Com isso, constata-se a necessidade de evitar o estresse intelectual, principalmente na hipótese de CQs geradas automaticamente.

Assim sendo, a motivação deste trabalho faz referência a:

- Prover alternativas para potencializar as ferramentas que buscam responder CQs que não foram validadas por uma ontologia, evitando a intervenção do engenheiro de ontologias e suspenção do processo de avaliação;

- O reuso de grandes ontologias de topo (e.g. SUMO) para aquisição de conhecimentos sobre conceitos gerais, visando beneficiar o maior número de ontologias de domínio desenvolvidas em ferramentas da área;

- Investigar as características de sistemas provadores automáticos de teoremas (ATP) e motores de inferências no que se refere a raciocínio automático sobre ontologias, buscando por soluções para minimizar as adversidades que possam ser encontradas;

- Buscar meios de facilitar e incentivar a utilização de uma possível solução para validação automática de CQs em ferramentas da engenharia de ontologias;

1.2 Objetivo Geral

O Objetivo Geral desta dissertação foi desenvolver um *framework*[13] capaz de utilizar a Ontologia de Topo SUMO em FOL e otimização de desempenho para manipular e validar CQs por raciocínio automático em um sistema provador automático de teoremas.

A este objetivo geral correspondem os objetivos específicos indicados a seguir:

1.2.1 Objetivos Específicos

1. Desenvolver um componente computacional capaz de manipular ontologias e CQs em FOL, fazendo uso de um sistema ATP para possibilitar a realização de inferências automáticas sobre a ontologia SUMO.

2. Disponibilizar um mecanismo de *cache*[14] capaz de armazenar validações concretizadas, objetivando evitar repetição de atividades em sistemas ATP por intemédio de uma ontologia OWL 2 DL e motores de inferência;

3. Implementar e demonstrar o uso alternativo de um método para selecionar premissas na base de conhecimento por representação e busca em grafo[15], tendo como objetivo diminuir o tempo do processo de inferência em sistemas ATP;

4. Possibilitar o uso de um dicionário léxico como alternativa no caso de uma CQ envolver conceitos desconhecidos, buscando encontrar o significado deles em referência aos conhecimentos que estão representados na SUMO;

5. Desenvolver uma arquitetura integrada por componentes com propósitos bem definidos para viabilizar as funcionalidades do *framework* pretendido, sobretudo permitindo que *CoreACQ* seja capaz de alcançar seu objetivo geral.

1.3 Principais contribuições

As principais contribuições deste trabalho são descritas a seguir:

1. A criação de um *framework* computacional para validar CQs através de conhecimentos que podem ser deduzidos pela SUMO, principalmente fazendo uso de um sistema ATP capaz de realizar raciocínio automático sobre ontologias em FOL;

[13] Em desenvolvimento de software, um framework é uma abstração que une funções comuns entre vários projetos de software provendo uma funcionalidade genérica

[14] Em computação, *cache* é entendido como uma área de armazenamento onde dados frequentemente usados são guardados para um acesso futuro mais rápido, poupando tempo e uso desnecessário do hardware

[15] Em computação, um grafo pode ser entendido como um conjunto de elementos unidos por arestas

2. A elaboração de uma arquitetura de referência, baseada em componentes, para o projeto de sistemas que propõem-se a manipular ontologias e axiomas em FOL, principalmente tendo como objetivo a aplicação de inferências;

3. O desenvolvimento de um procedimento para reduzir o tempo do processo de raciocínio em um sistema ATP, para isso, implementando uma função de seleção de premissas por representação e busca de axiomas em grafos.

4. O incentivo ao reuso de informações semanticamente definidas por ontologias de topo livres e gratuitas, como a SUMO, posto que a definição formal de conceitos e documentos na *Web* não é uma tarefa trivial e requer profissionais da engenharia de ontologias;

1.4 Visão Geral da Solução Proposta

Este trabalho tem como finalidade o desenvolvimento do *CoreACQ*, um *framework* para validação de CQs através de informações resultantes de raciocínio automático sobre a SUMO. Assim, um engenheiro pode ser favorecido com a descoberta automática de conhecimentos relevantes que precisam ser adquiridos para avaliação de sua ontologia de domínio. Esta seção descreve, de uma forma sucinta, as características e funcionalidades do *framework* em questão.

CoreACQ foi implementado através de uma arquitetura que combina quatro módulos principais, são eles: (1) *Core ATP*; (2) *Core OWL-DL*; (3) *Preprocessor*; e (4) *SUMO Inference Engine*. De modo geral, o primeiro (1) é responsável pela comunicação com sistemas ATP. O segundo (2) disponibiliza recursos necessários para a manipulação de uma ontologia OWL 2 DL que se refere ao mecanismo de *cache*. O terceiro (3) responsabiliza-se por carregar em memória[16] uma ontologia da SUMO em FOL, interagir com o dicionário léxico *WordNet* para identificação de conceitos e, também, realizar seleção de premissas por representação e busca de axiomas em grafos. O útlimo (4) incumbe-se por manipular e validar CQs por meio de consultas, entretando fazendo uso de funções executadas pelos demais módulos. Isto posto, os módulos e seus respectivos componentes foram desenvolvidos tendo em vista uma alta coesão[17] e um baixo acoplamento[18] para facilitar manutenções futuras.

A Figura 1, a seguir, apresenta um fluxograma exemplificando o funcionamento da arquitetura do *CoreACQ*. Primeiro (1) faz-se necessário informar qual a versão da SUMO em FOL será utilizada. Em seguida (2), ocorre a inicialização do sistema ATP (e.g. *E*

[16] Em informática, memória refere-se aos dispositivos que permitem ao computador guardar dados, seja temporariamente ou permanentemente

[17] Em engenharia de *software*, coesão pode ser entendida como uma medida para avaliar se todas as funcionalidades de um componente são de sua responsabilidade

[18] Em engenharia de *software*, acoplamento refere-se ao grau de dependência de um componente com relação a outros para exercer suas funções

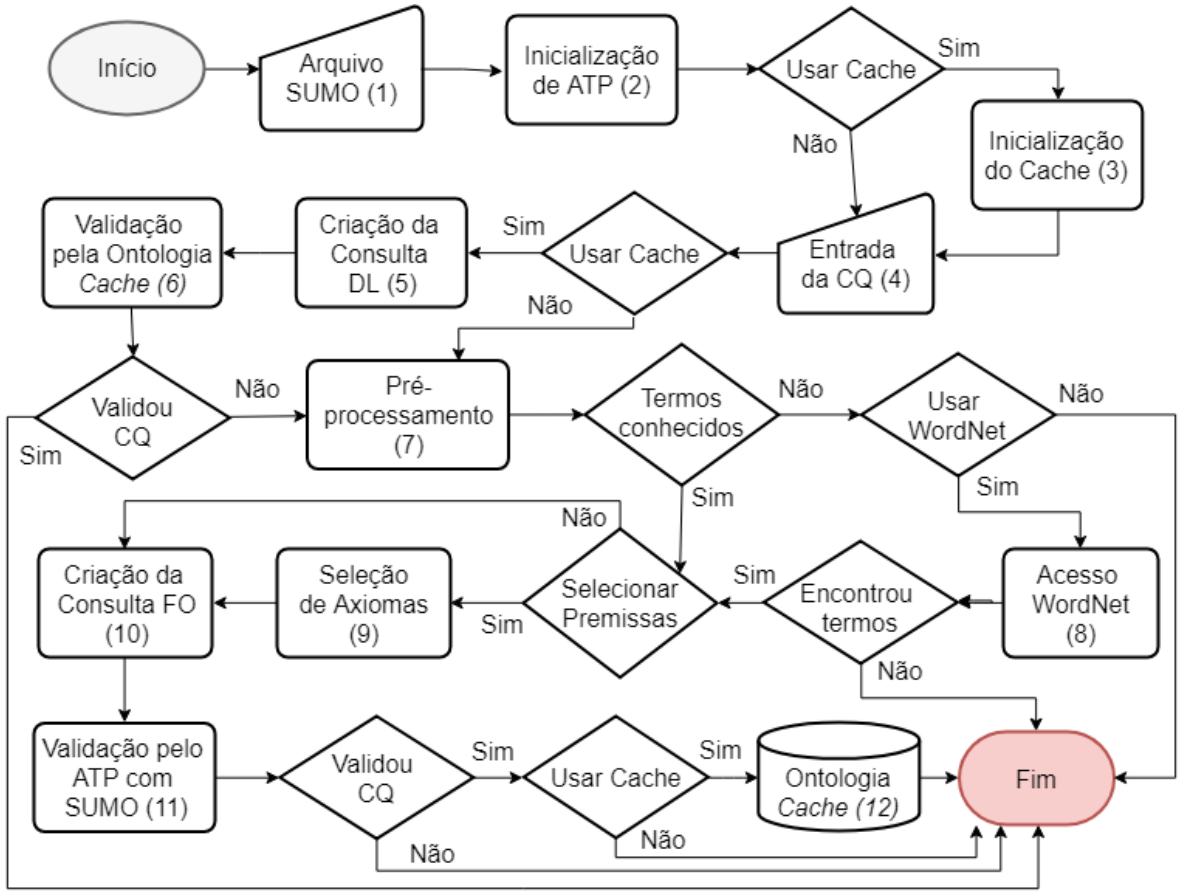

Figura 1 – Fluxograma de execução do *CoreACQ*

Prover), responsável por realizar inferências sobre axiomas em FOL. Neste momento é preciso decidir o uso ou não da *Cache*, caso sim o mecanismo também será inicializado (3). Logo após, uma CQ pode ser recebida como entrada (4) na forma de uma consulta. Considerando-se o uso da *Cache*, a consulta é representada em DL (5) e sua validação é processada por tal mecanismo (6). Na hipótese da CQ ser validada - isto é, deduzida por intermédio da ontologia *Cache* - o *framework* encerra a operação e informa o respectivo resultado. Caso contrário, acontece o pré-processamento (7) da consulta com o intuito de identificar conceitos desconhecidos (8) e selecionar premissas da base de conhecimento pelos conceitos da CQ (9). Depois disso, a consulta é representada em FOL (10) e sua validação é processada pelo sistema ATP com a SUMO (11). Se a CQ for validada, então essa informação é armazenada na *Cache* (12) e, posteriormente, fornecida como resposta.

1.5 Escopo e Limitações

Os seguintes pontos não são abordados neste trabalho:

- **Criação de *Graphical User Interface* (GUI, em tradução livre Interface Gráfica do Usuário)**. *CoreACQ* é um *framework* computacional para validação

de CQs com o objetivo de ser interligado às ferramentas existentes da engenharia de ontologias. Portanto, este trabalho interage indiretamente com usuários finais (e.g. engenheiros) e não pretende criar interfaces gráficas.

- **Demonstrar metodologias e padrões da engenharia de ontologias**. Está fora do escopo deste trabalho demonstrar o uso de metodologias, padrões ou boas práticas voltadas ao desenvolvimento de ontologias. Sobre esses tópicos, *CoreACQ* dispõe-se apenas a promover a utilização de CQs no processo de avaliação e evolução de ontologias de domínio.

- **Métodos para geração automática de CQs**. *CoreACQ* propõe-se a manipular CQs previamente definidas, seja de forma manual por um engenheiro de ontologias ou automática por ferramentas de terceiros. Portanto, está fora do escopo deste trabalho abordar ou dispor métodos para geração automática de CQs.

1.6 Organização do texto

O desenvolvimento deste trabalho está definido em 6 capítulos e está organizado da seguinte maneira:

- No Capítulo 2 é apresentada a fundamentação teórica necessária para entender este trabalho, fornecendo uma visão geral sobre o conceito de ontologias, bem como linguagens para representação de conhecimento e raciocínio automático sobre KIF, FOL e DL. Ainda neste capítulo são abordados aspectos a respeito de algoritmos de busca em grafos.

- O Capítulo 3 apresenta a arquitetura e as funcionalidades do *CoreACQ*, detalhando cada módulo e componente do *framework*, principalmente no tocante a validação de CQs através de consultas e raiocioínio automático sobre a SUMO, mecanismo de *cache* por ontologia OWL 2 DL, comunicação com a *WordNet* e, também, seleção de premissas por representação e busca de axiomas em grafos.

- No Capítulo 4, por sua vez, é apresentada a metodologia adotada para a avaliação do *framework* proposto, mostrando todos os testes realizados para legitimar a capacidade do *CoreACQ* em atingir seu objetivo geral e os objetivos específicos definidos.

- No Capítulo 5 são apresentados os trabalhos relacionados, a descrição de cada um deles e, ainda, suas características e diferenças em comparação com *CoreACQ*.

- No Capítulo 6 são abordadas as conclusões desta dissertação, abrangendo suas contribuições, competências, limitações e expectativas de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais conceitos envolvidos na pesquisa, objetivando sobretudo fornecer uma percepção ampliada sobre os conceitos de Ontologias, Raciocínio Automático sobre Ontologias, a Ontologia de Topo SUMO, e um algoritmo para resolução de problemas por meio de busca, denimonado Busca em Largura.

2.1 Ontologias

Os conteúdos da *Web* convencional são projetados para serem entendidos por humanos, não para programas de computador - também chamados de agentes de software - manipularem o seu significado. Os computadores são capazes de analisar as páginas (sites) para exibição e rotinas de processamento - sobretudo ligações (*links*) de redirecionamento entre páginas - mas em geral, computadores não tem habilidade para processar a semântica de documentos da *Web* de forma autônoma, por exemplo: indicar que uma página é sobre Alan Turing e recomendar uma ligação que contenha informações sobre sua bibliografia. A *Web* Semântica objetiva justamente estruturar o significado do conteúdo das páginas, criando um ambiente onde programas de computador navegando de uma página para outra podem raciocinar e realizar tarefas sofisticadas para o usuário (BERNERS-LEE; HENDLER; LASSILA, 2001).

Uma solução para definir o significado de páginas *Web* é proporcionada pela *Web* Semântica através de coleções de informações chamadas de ontologias. A filosofia indica uma ontologia como uma teoria sobre a natureza do ser ou existência, entretando, conforme Berners-Lee, Hendler e Lassila (2001), em Inteligência Artificial (IA, em inglês *Artificial Intelligence*) uma ontologia pode ser definida como um documento ou arquivo que define formalmente as relações entre os termos (conceitos). Outra definição mais formal seria "o conjunto de entidades com suas relações, restrições, axiomas e um vocabulário" (FREITAS, 2003).

No contexto da *Web* Semântica, as ontologias descrevem fatos e entidades acerca de um determinado domínio de conhecimento, possuindo uma hierarquia de conceitos (ou taxonomia) e um conjunto de regras de inferência que pode ser usado para conduzir o processo de raciocínio automático. Ademais, o desenvolvimento de uma ontologia fundamenta-se nos seguintes componentes básicos:

- **Indivíduos (ou instâncias)**: representam objetos do mundo, por exemplo, *Alan Turing* e *Ethel Sara*.
- **Classes**: são conjuntos de instâncias, por exemplo, as classe *Man* (*Homem*) e *Woman* (*Mulher*) que contêm, respectivamente, *Alan Turing* e *Ethel Sara* como indiví-

duos. Geralmente, as classes devem ser organizadas de acordo com uma hierarquia, por exemplo, *Man* e *Woman* podem ser definidas como subclasses - isto é, subconjuntos - da classe *Human* (*Humano*), assim, por raciocínio de subsunção, *Alan Turing* e *Ethel Sara* também são classificados como indivíduos de *Human*.

- **Propriedades**: representam relações entre instâncias, por exemplo, a relação de *parentesco* entre duas pessoas de uma mesma família.
- **Axiomas**: são sentenças (afirmações) consideradas sempre verdadeiras, por exemplo, *Alan Turing é parente de Ethel Sara*.

Na Figura 2, a seguir, é apresentado um exemplo simples de uma ontologia e seus componentes. A classe *Man* possui dois indivíduos, *Julius Turing* e *Alan Turing*. Já a classe *Woman* tem apenas um indivíduo, *Ethel Sara*. Ainda é possível observar uma hierarquia de classes, onde a classe *Human* é superclasse - isto é, contém todos os indivíduos - de *Man* e *Woman*, ou seja, possui os indivíduos *Alan Turing*, *Julius Turing* e *Ethel Sara*. Observa-se também a existência da propriedade *relativeOf*, cujo significado indica um relacionamento entre dois indivíduos de uma mesma família. Por fim, encontram-se dois axiomas, o primeiro determina que *Alan Turing* é parente de *Julius Turing*, do mesmo modo o segundo aponta que *Alan Turing* é parente de *Ethel Sara*.

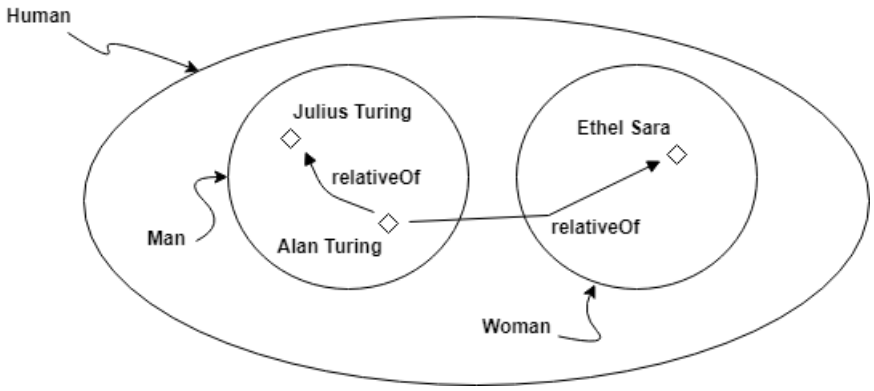

Figura 2 – Exemplo de representação dos componentes básicos de uma ontologia

Embora as ontologias compartilhem estruturas semelhantes, elas podem ser categorizadas de acordo com seu grau de generalização. Freitas (2003) lista os diferentes tipos de ontologias em oderm decrescente de genericidade[1], da mais genérica até a mais específica, conforme a seguir:

- **Ontologias de representação**: definem as primitivas de representação - como axiomas, atributos e outros - de forma declarativa.

[1] Genericidade pode ser definida como a condição daquilo que é genérico

- **Ontologias gerais (ou de topo)**: trazem definições abstratas de propósito geral necessárias para a compreensão de características do mundo, como tempo, processos, papéis, espaços, seres, coisas, etc.
- **Ontologias centrais (ou de nível médio)**: determinam os ramos de estudo e/ou conceitos genéricos e abstratos de uma área. Por exemplo, uma ontologia central de direito pode incluir conhecimentos normativos, de responsabilidade, reativos, de agências legais, comportamentos permitidos, etc. Assim, tais conceitos e conhecimentos servem de base para ontologias de ramos mais específicos do direito, como direito tributário, de família e outros.
- **Ontologias de domínio**: abordam conceitos de um domínio mais específico de uma área genérica de conhecimento, como direito tributário, microbiologia, etc.
- **Ontologias de aplicação**: buscam solucionar problemas específicos sobre um domínio de conhecimento, como identificar doenças do coração fazendo uso de uma ontologia de domínio sobre cardiologia.

Para exemplificar, é apresentada na Figura 3 uma hierarquia de classes dividida em diferentes tipos de ontologias. A ontologia de topo (*Upper*) possui a superclasse *Thing* (Coisa) e suas respectivas subclasses, *Artifact* (Artefato) e *Organism* (Organismo). A ontologia central (*Mid-level*) contém cinco classes, *Animal* (Animal), *Worm* (Minhoca), *Human* (Human), *Man* (Homem) e *Woman* (Mulher), respeitando a seguinte hierarquia: *Animal* é subclasse de *Organism*; *Worm* e *Human* são subclasses de *Animal*; *Man* e *Woman* são subclasses de *Human*. Já a ontologia de domínio (*Domain*) tem duas classes, *Actor* (Ator) definida como subclasse de *Man*, e *Actress* (Atriz) como subclasse de *Woman*. Por fim, a ontologia de aplicação (*Application*) possui dois indivíduos, *Will Smith* como uma instância de *Actor*, e *Julia Roberts* como uma instância de *Actress*.

A subdivisão de ontologias em tipos beneficia o reuso e compartilhamento de conhecimento entre pessoas e agentes de software. Freitas (2003) destaca os benefícios das ontologias por manifestar que elas proporcionam, por exemplo:

- A possibilidade para desenvolvedores de reusar ontologias, uma vez que a construção de bases de conhecimento é uma tarefa cara.
- A existência de uma gama variada de "ontologias de prateleira", disponíveis para uso e reuso por pessoas e agentes.
- A perspectiva de tradução entre várias linguagens e formalismos de representação de conhecimento.
- O acesso a servidores de ontologias via Internet, capazes de armazenar grandes massas de classes e indivíduos, favorecendo empresas e grupos de pesquisas que podem cooperar para manter a integridade do conhecimento compartilhado.

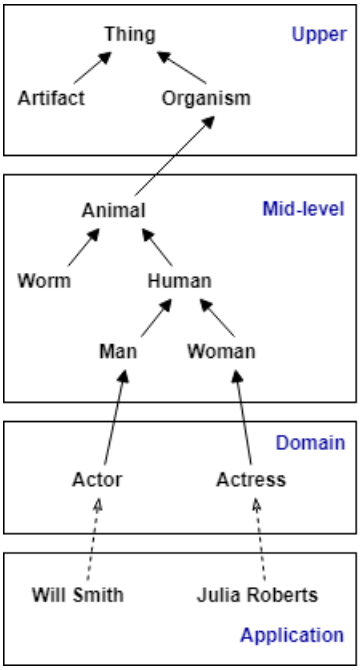

Figura 3 – Exemplo de ontologias separadas de acordo com seus tipos

Na subseção 2.1.1, a seguir, são abordadas algumas linguagens para representação de conhecimento usadas na construção de ontologias, evidenciando suas principais características, vantagens e limitações.

2.1.1 Linguagens de Representação de Conhecimento

Nesta subseção são apresentadas quatro linguagens para representação de conhecimento, a primeira referente a SUO-KIF, a segunda a respeito de FOL, a terceira sobre DL e a última com respeito a OWL 2 DL.

2.1.1.1 *SUO-KIF*

Standard Upper Ontology - Kownledge Interchange Format (SUO-KIF, em tradução livre "Formato para Troca de Conhecimento da Ontologia de Topo Padrão") é uma linguagem desenvolvida para criação e troca de conhecimentos, derivada de KIF (*Konwledge Interchange Format*, em tradução livre Formato para Troca de Conhecimento), que é uma linguagem de alta ordem[2] logicamente abrangente - isto é, proporciona a expressão de sentenças com expressividade superior à FOL. Todavia, SUO-KIF destina-se a ser o mais próximo possível de uma linguagem de primeira ordem (PEASE, 2004).

Os termos (ou conceitos) em SUO-KIF podem referir-se a indivíduos como *Will Smith*, ou classes como *Ator* e *Atriz*. Do mesmo modo, é possível definir relações (propriedades) como *parente* e funções como *multiplicação*. Contudo as relações e funções são instâncias

[2] Lógica de segunda ordem, ou de alta ordem, é mais expressiva que a lógica de primeira ordem.

de classes acerca de, respectivamente, relações e funções. Visando facilitar a compreensão dos termos, ontologias como a SUMO denominam funções por meio do sufixo "*Fn*" (e.g. *MultiplicationFn*) e relações pela primeira letra minúscula (e.g. *relativeOf*).

Os termos são combinados em sentenças com o objetivo de criar axiomas, por exemplo, o axioma "*Will Smith* é um ator" pode ser representado em SUO-KIF como:

```
(instance WillSmith Actor)
```
(2.1)

SUO-KIF também permite a conexão de sentenças com os conectivos lógicos de conjunção (*and*) e disjunção (*or*). Por exemplo, o axioma "*Julia Roberts* é uma mulher e venceu o Óscar de melhor atriz" pode ser representado como:

```
(and
    (instance JuliaRoberts Woman)
    (hasWon JuliaRoberts Óscar BestActress)
)
```
(2.2)

Sentenças ainda podem ser negadas através do conectivo lógico de negação (*not*). Para exemplificar, o axioma "*Will Smith* não é parente de Julia Roberts" pode ser representado em SUO-KIF conforme a seguir:

```
(not
    (relative WillSmith JuliaRoberts)
)
```
(2.3)

SUO-KIF também suporta funções, conforme Pease (2004) "uma função é uma relação que para todo valor de um domínio tem apenas um único valor na imagem". No entanto, esta linguagem permite expressões funcionais que denotam termos, por exemplo, ***(GovernmentFn Brazil)*** resulta um indivíduo que é o presidente do Brasil.

O conector lógico de implicação é donatado em SUO-KIF pelo símbolo =>, que pode ser entendido como *implica*, ou "*se* argumento-1 *então* argumento-2". Os termos são combinados para formar axiomas como "*Se* um indivíduo (?P) é uma pessoa *então* possui como característica sexo masculino ou feminimo", como visto no exemplo a seguir:

```
(=>
    (instance ?P Human)
    (or
        (attribute ?P Male)
        (attribute ?P Female)
    )
)
```
(2.4)

O conector lógico de bi-implicação é denotado em SUO-KIF pelo símbolo <=>, que é uma forma abreviada para a combinação de duas implicações, formalmente, o axioma *(<=> **A B)*** é equivalente a ***(and (=> A B) (=> B A))***.

Outros dois operadores lógicos do SUO-KIF são ***exists*** e ***forall***. O quantificador existencial, ***exists***, é utilizado para indicar a existência de pelo menos um termo (indivíduo ou relação) que satisfaz uma determinada sentença. Para exemplificar, o axioma "*Se* um indivíduo (?P) é uma pessoa *então existe* pelo menos uma pessoa que é seu parente (?R)" pode ser representado como:

```
(=>
    (instance ?P Human)
    (exists ?R
        (and
            (instance ?R Human)
            (relativeOf ?P ?R)
        )
    )
)
```
(2.5)

O quantificador universal, ***forall***, especifica que uma sentença é válida para todas as entidades (indivíduos ou relações) que satisfazem suas variáveis (?), por exemplo, o axioma "Todos os parentes (*?P* e *?R*) são pessoas e se gostam" é declarado conforme a seguir:

```
(forall ?P ?R
    (=>
        (relativeOf ?P ?R)
        (and
            (instance ?P Human)
            (instance ?R Human)
            (likes ?P ?R)
        )
    )
)
```
(2.6)

A forte expressividade da SUO-KIF pode ser constatada em sentenças que predicam sobre relações, isto é, possuem pelo menos uma variável com referência a uma relação. Por exemplo, "*Se* a relação *?R* é simétrica *então* para quaisquer dois indivíduos *?I1* e *?I2* é verdade que o relacionamento *(?R ?I1 ?I2) implica (?R ?I2 ?I1)*", segundo a Fórmula 2.7, a seguir. Em contrapartida, isso eleva a complexidade do projeto de um raciocinador automático, como um provador de teoremas para linguagem de alta ordem.

```
(=>
    (instance ?R SymmetricRelation)
    (forall (?I1 ?I2)
        (=>
            (?R ?I1 ?I2)
            (?R ?I2 ?I1)
        )
    )
)
```
(2.7)

Na Subseção 2.1.1.2, a seguir, é apresentada outra linguagem de representação de conhecimento conhecida como FOL.

2.1.1.2 *First-Order Logic (FOL)*

A lógica proposicional é muito simples pois ela não possui a capacidade de expressão necessária para lidar com relacionamentos entre objetos (e.g. indivíduos e classes) de forma concisa. *First-Order Logic* (FOL, em tradução Lógica de Primeira Ordem) - também

conhecido como lógica de predicados - surge como um sistema lógico formal que estende a lógica proposicional por oferecer os operadores de quantificação existencial e universal. Russell e Norvig (2009) complementam ainda:

> A lógica de primeira ordem é suficientemente expressiva para representar de forma satisfatória nosso conhecimento comum. Ela também compõe ou forma os alicerces de muitas outras linguagens de representação e foi intensivamente estudada por muitas décadas.

Os elementos básicos de FOL são os símbolos que representam os objetos, relações, e funções. Os símbolos, portanto, são divididos em três tipos: **símbolos de constantes**, que representam objetos; **símbolos de predicados**, que representam relações (propriedades); e **símbolos de funções**, que representam funções. Visando simplificar a distinção dos símbolos, ontologias como a *AdimenSUMO* identificam classes pelo prefixo "c_" (e.g. *c_Actor*), indivíduos pela primeira letra maiúscula (e.g. *WillSmith*) e funções pelo prefixo "f_" (e.g. *f_Multiplication*).

Um modelo em FOL consiste de um conjunto de objetos e uma interpretação que mapeia símbolos constantes para objetos, símbolos de predicados para relações entre os objetos, e símbolos de funções para funções. Um modelo deve fornecer as informações necessárias para determinar se alguma sentença é verdadeira ou falsa, portanto, deve compreender uma interpretação que especifica exatamente quais objetos, relações e funções são referidos pelos símbolos de constantes, predicados e funções, conforme os exemplos a seguir:

- *WillSmith* refere-se ao indivíduo Will Smith e *JadenSmith* diz respeito ao indivíduo Jaden Smith.
- *Woman* (em tradução livre, Mulher) refere-se à classe de todos os indivíduos que são mulheres.
- *father* (em tradução livre, pai) refere-se à relação de paternidade, isto é, um indivíduo que tem como propriedade ser pai de outro indivíduo.
- *Multiplication* (em tradução livre, Multiplicação) refere-se a função de multiplicação entre dois ou mais números.

Um termo é uma expressão lógica que se refere a um objeto. Símbolos constantes são termos, entretanto, geralmente não é conveniente ter um símbolo distinto para nomear todos os objetos. Por exemplo, podemos usar a expressão "Will Smith's head" (em tradução livre, "a cabeça de Will Smith") para referenciar à cabeça de Will Smith sem a necessidade de nomeá-la. Os símbolos de funções são usados para, por exemplo, permitir

o uso de *head(WillSmith)* em vez de um símbolo constante. Dessa forma, o termo de função $f(t_1,...,t_n)$ é nomeado por f, possui aridade[3] n e seus termos de argumentos $(t_1,...,t_n)$ referem-se a objetos ou variáveis.

Em FOL, uma sentença atômica é formada a partir de um símbolo de predicado seguido por uma lista de n termos entre parênteses, onde n é chamado de aridade do predicado. Para exemplificar, as sentenças atômicas "Will Smith é um homem" e "Will Smith é pai de Jaden Smith" podem ser representadas como visto a seguir:

$$Man(WillSmith) \tag{2.8}$$

$$father(WillSmith,\ JadenSmith) \tag{2.9}$$

Os conectivos lógicos fazem parte da definição de FOL, eles são usados para formar sentenças complexas a respeito de sentenças atômicas. A conjunção ($\wedge$) de duas sentenças S_1 e S_2 é uma nova sentença que é verdadeira se e somente se S_1 e S_2 forem verdadeiras. Já a disjunção ($\vee$) é uma nova sentença que é verdadeira quando S_1 ou S_2 for verdadeira. Para exemplificar, o axioma "*Julia Roberts* é uma mulher e venceu o Óscar de melhor atriz" pode ser representado conforme a seguir:

$$Woman(JuliaRoberts) \wedge Oscar(BestActress) \wedge hasWon(JuliaRoberts,\ BestActress) \tag{2.10}$$

Em FOL, a negação ($\neg$) de uma sentença S_1 é uma nova sentença cujo valor verdade[4] é o oposto do valor verdade de S_1, por exemplo, o axioma "*Will Smith* não é parente de *Julia Roberts*" pode ser representado conforme a seguir:

$$\neg \quad relativeOf(WillSmith,\ JuliaRoberts) \tag{2.11}$$

Em FOL, a implicação ($\rightarrow$) entre duas sentenças S_1 e S_2 é uma nova sentença que representa o condicional "Se S_1 é verdadeira então S_2 é verdadeira". Para exemplificar, predicados são combinados para formar axiomas como "*Se* um indivíduo (x) é um humano *então* possui como característica sexo masculino ou feminino", como visto a seguir:

$$Human(x) \ \rightarrow \ (\ Male(x) \ \vee Female(x)\ ) \tag{2.12}$$

O conectivo lógico de bi-implicação em FOL é denotado pelo símbolo $\leftrightarrow$, que é uma abreviação para a conjunção de duas implicações, formalmente, $\boldsymbol{A} \leftrightarrow \boldsymbol{B}$ é o mesmo que $\boldsymbol{A} \rightarrow \boldsymbol{B} \wedge \boldsymbol{B} \rightarrow \boldsymbol{A}$.

[3] Em lógica de primeira ordem, a aridade determina o número de argumentos de um símbolo de função ou predicado.

[4] Em lógica de primeira ordem, valor verdade, também chamado de valor de verdade, é um valor que indica se uma sentença é verdadeira ou falsa.

Os operadores de quantificação em FOL são existencial (∃) e universal (∀). O existencial indica que existe algum indivíduo - denotado por uma variável - que satisfaz determinada sentença. Para exemplificar, o axioma "*Se* um indivíduo (x) é um humano *então* existe pelo menos um humano que é seu parente (y)" pode ser representado como:

$$Human(x) \rightarrow \ \exists y \ (\ Human(y) \wedge relativeOf(x,\ y)\ ) \tag{2.13}$$

A Fórmula 2.13, apesar de correta, não é considerada uma sentença em FOL devido ao uso da variável livre x, ou seja, toda variável de uma sentença em FOL deve estar ligada a algum quantificador. O quantificador universal (∀) deve ser usado quando uma certa sentença é válida para todas as entidades (indivíduos ou classes) que satisfazem suas variáveis. Para exemplificar, a Fórmula 2.13 pode ser reescrita com o propósito de formar o axioma "Para todo humano existe pelo menos um parente que também é um humano", conforme a seguir:

$$\forall x \ (\ Human(x) \rightarrow \ \exists y \ (\ Human(y) \wedge relativeOf(x,\ y)\ )\quad) \tag{2.14}$$

É apresentado na Subseção 2.1.1.3, a seguir, a linguagem de representação de conhecimento denominada DL.

2.1.1.3 *Description Logic (DL)*

Description logics (DL, em tradução livre Lógica de Descrições) é uma família de linguagens que pode ser usada para representar o conhecimento de uma aplicação de domínio de uma forma estruturada e formalmente compreendida. Uma motivação para o nome *lógica de descrições* refere-se ao fato de que as noções importantes sobre um domínio são definidas por descrições conceituais, isto é, expressões construídas a partir de conceitos atômicos (predicados unários[5]) e axiomas atômicos (predicados binários[6]) por intermédio de construtores fornecidos pela DL (BAADER; HORROCKS; SATTLER, 2008).

Classes em DL são representadas por meio de símbolos, como *Actor* e *Actress* que contêm, respectivamente, indivíduos atores e atrizes. Existem vários construtores de axiomas aplicados sobre classes, como, por exemplo, união (⊔) e interseção (⊓). A união é entendida como o conectivo lógico de disjunção, já a interseção corresponde à conjunção lógica. Para exemplificar, o axioma "A classe de todos os indivíduos que são atores ou atrizes" pode ser representado conforme a seguir:

$$Actor \quad \sqcup \quad Actress \tag{2.15}$$

[5] Predicado unário é um predicado com apenas um argumento.
[6] Predicado binário é entendido como um predicado com exatamente dois argmentos.

Outro construtor em DL é conhecido como complemento ($\neg$), cujo significado corresponde ao conectivo lógico de negação, por exemplo, o axioma "A classe de todos os indivíduos que não são atores ou atrizes" pode ser representado como:

$$\neg\ (Actor \quad \sqcup \quad Actress) \tag{2.16}$$

Relacionamentos (ou propriedades) em DL são tratados através de predicados, da mesma forma que ocorre em FOL, entretanto, aqui toda relação tem exatamente dois argumentos, isto é, podemos relacionar apenas dois indivíduos por vez. Diante disso, encontram-se outros três construtores conhecidos como restrição existencial ($\exists R.C$), restrição de valor ($\forall R.C$) e restrição de número ($\geq\ n\ R$). Para exemplificar, o axioma "a classe de atores que estrelaram em apenas filmes, ganharam mais de um prêmio e pelo menos um Globo de Ouro" pode ser representado conforme a seguir:

$$Actor\ \sqcap\ \forall hasFilmed.Movie\ \sqcap\ (\geq 2\ hasWon.Award)\ \sqcap\ \exists hasWon.GoldenGlobe \tag{2.17}$$

Descrições de conceitos podem ser usadas para construir afirmações em uma base de conhecimento DL, tipicamente dividida em duas partes: uma terminológica e uma de asserções (ou instâncias). Na parte *terminológica*, denominada TBox, são descritas noções relevantes a cerca de conceitos e relacionamentos entre eles - corresponde ao esquema de um banco de dados. Um tipo de declaração (ou axioma) TBox pode definir um nome (abreviação) para uma descrição complexa mediante o construtor de equivalência ($\equiv$), por exemplo, o nome *AwardedActor* como uma abreviação para a classe definida em 2.17:

$$\begin{aligned} AwardedActor \equiv\ & Actor\ \sqcap\ \forall hasFilmed.Movie\ \sqcap\ (\geq 2\ hasWon.Award)\ \sqcap \\ & \exists hasWon.GoldenGlobe \end{aligned} \tag{2.18}$$

A hierarquia de conceitos em uma ontologia DL é estabelecida por intermédio de declarações TBox, especificamente pelo construtor de subsunção ($\sqsubseteq$), por exemplo, os axiomas "um ator é uma pessoa, uma pessoa é um mamífero e todo mamífero é um animal" podem ser representados da seguinte maneira:

$$Mammal \sqsubseteq\ Animal \tag{2.19}$$

$$Person \sqsubseteq\ Mammal \tag{2.20}$$

$$Actor \sqsubseteq\ Person \tag{2.21}$$

Formalismos terminológicos mais expressivos permitem ainda axiomas como "todo ator ou atriz estrelou em pelo menos uma série de TV ou um filme":

$$Actor \sqcup Actress \sqsubseteq \exists hasFilmed.(Movie \sqcup TelevisionSeries) \tag{2.22}$$

A parte de *asserções* de uma base de conhecimento, chamada ABox, é utilizada para descrever uma situação concreta ao declarar características de indivíduos - corresponde aos dados em um banco de dados. Para exemplificar, o axioma "Julia Roberts é uma atriz e ganhou o Óscar de melhor atriz" pode ser representado de acordo com os seguintes formalismos assertivos:

$$Actress(JuliaRoberts) \tag{2.23}$$

$$hasWon(JuliaRoberts, \ \ OscarsBestActress) \tag{2.24}$$

Outro recurso importante para uma linguagem de ontologia envolve a noção de propriedade inversa ($^-$), assim, em DL é possível a criação de declarações do tipo "um pai de um humano também deve ser um humano", conforme a seguir:

$$\exists hasParent^- .Human \sqsubseteq Human \tag{2.25}$$

No exemplo da Fórmula 2.25, acima, a propriedade $hasParent^-$ possui significado semelhante ao de *hasChild*, que por sua vez pode ser definida como propriedade inversa de *hasParent*.

Todo o conhecimento em DL descrito até então é facilmente representado em FOL. A sintaxe livre de variáveis da DL torna as declarações TBox mais fáceis de compreender do que as fórmulas correspondentes em FOL. Contudo, a principal razão para usar DL diz respeito a sua cuidadosa adaptação para proporcionar expressividade suficiente com a decidibilidade[7] de problemas significativos de raciocínio relacionado a ontologias.

Na Subseção 2.1.1.4, a seguir, é apresentada a linguagem de representação de conhecimento conhecida como OWL 2.

2.1.1.4 *Ontology Web Language 2 - OWL 2*

A *Web Ontology Language 2* (em tradução livre Linguagem de Ontologia para Web), informalmente conhecida como OWL 2, é uma linguagem de representação de conhecimento que foi desenvolvida pelo *W3C OWL Working Group* (em tradução livre, W3C Grupo de Trabalho da OWL). OWL 2 tem como maior objetivo facilitar a construção e o compartilhamento de ontologias através da *Web*, proporcionando classes, propriedades, indivíduos e valores de dados que são armazenados como documentos da Web Semântica (W3C OWL Working Group, 2012).

[7] Em lógica, o termo decidível refere-se a um problema de decisão, isto é, a questão da existência de um método capaz de determinar a legitimidade em um conjunto de fórmulas.

A atribuição de significado em ontologias OWL 2 pode ser feita mediante a *OWL 2 Direct Semantics* (em tradução livre, Semântica Direta da OWL 2), que atribui significado diretamente às estruturas ontológicas. Semântica direta é compatível com a lógica de descrição conhecida como SROIQ, cuja expressividade é mais poderosa do que os construtores DL discutidos na Seção 2.1.1.3. À vista disso, *W3C OWL Working Group (2012)* estabelece "OWL 2 DL" como nomenclatura informal para referenciar ontologias OWL 2 interpretadas por semântica direta.

Ontologias OWL 2 e seus elementos são identificados por intermédio de *Internationalized Resource Identifiers* (IRI, em tradução livre Idetificadores de Recursos Internacionalizados). IRI são identificadores absolutos, isto é, dois IRIs são estruturalmente equivalentes se e somente se suas representações em texto são idênticas, por exemplo, a IRI "*http://cinema.edu#Actor*" refere-se a classe de todos os indivíduos que são atores. Assim sendo, o esqueleto básico de uma ontologia OWL 2 DL caracteriza informações sobre suas próprias estruturas e sua IRI, conforme demonstrado na Figura 4.

```
<rdf:RDF xmlns="http://cinema.edu#"
     xml:base="http://cinema.edu"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="http://cinema.edu"/>

    <!--
    ///////////////////////////////////////////////////////
    //
    // Classes, propriedades, indivíduos e relacionamentos
    // são definidos entre <rdf:RDF> e <rdf:/RDF>
    //
    ///////////////////////////////////////////////////////
     -->

</rdf:RDF>
```

Figura 4 – Exemplo da estrutura básica de uma ontologia OWL 2 DL na sintaxe RDF/XML

Em OWL 2 DL, classes e propriedades (ou relações) são distinguidas diretamente pela própria linguagem, por exemplo, a declaração da classe de Óscars e a declaração da propriedade *hasWon* - isto é, alguém ganhou algo - são apresentadas na Figura 5.

O construtor OWL 2 DL de subclasse (*rdfs:subClassOf*) corresponde ao construtor DL de subsunção ($\sqsubseteq$), para exemplificar, os axiomas "uma atriz é uma pessoa, uma pessoa

```
<owl:Class rdf:about="http://cinema.edu#Oscar"/>

<owl:ObjectProperty rdf:about="http://cinema.edu#hasWon"/>
```

Figura 5 – Exemplo de declarações de classe e propriedade em uma ontologia OWL 2 DL na sintaxe RDF/XML

é um mamífero e todo mamífero é um animal" podem ser representados como visto na Figura 6, a seguir:

```
<owl:Class rdf:about="http://cinema.edu#Actress">
    <rdfs:subClassOf rdf:resource="http://cinema.edu#Person"/>
</owl:Class>

<owl:Class rdf:about="http://cinema.edu#Person">
    <rdfs:subClassOf rdf:resource="http://cinema.edu#Mammal"/>
</owl:Class>

<owl:Class rdf:about="http://cinema.edu#Mammal">
    <rdfs:subClassOf rdf:resource="http://cinema.edu#Animal"/>
</owl:Class>
```

Figura 6 – Exemplo de declarações a respeito do construtor de subclasse em uma ontologia OWL 2 DL na sintaxe RDF/XML

Os construtores OWL 2 DL conhecidos como *owl:unionOf* e *owl:intersectionOf* correspondem, respectivamente, aos construtores DL de união ($\sqcup$) e interseção ($\sqcap$), por exemplo, o axioma "todos os atores são pessoas e possuem o sexo masculino" pode ser representado conforme observado na Figura 7, a seguir.

```
<owl:Class rdf:about="http://cinema.edu#Actor">
 <rdfs:subClassOf>
  <owl:Class>
    <owl:intersectionOf rdf:parseType="Collection">
     <rdf:Description rdf:about="http://cinema.edu#Male"/>
    <rdf:Description rdf:about="http://cinema.edu#Person"/>
   </owl:intersectionOf>
  </owl:Class>
 </rdfs:subClassOf>
</owl:Class>
```

Figura 7 – Exemplo de declaração a respeito do construtor de interseção em uma ontologia OWL 2 DL na sintaxe RDF/XML

Em relação ao construtor DL de complemento ($\neg$), ele é traduzido para o construtor OWL 2 DL denominado *owl:complementOf*, para exemplificar, o axioma "um indivíduo

que é ator não pode pertencer à classe de atrizes" pode ser representado conforme visto na Figura 8, a seguir.

```
<owl:Class rdf:about="http://cinema.edu#Actor">
 <rdfs:subClassOf>
  <owl:Class>
   <owl:complementOf rdf:resource="http://cinema.edu#Actress"/>
  </owl:Class>
 </rdfs:subClassOf>
</owl:Class>
```

Figura 8 – Exemplo de declaração a respeito do construtor de complemento em uma ontologia OWL 2 DL na sintaxe RDF/XML

Os construtores OWL 2 DL de restrição existencial (owl:someValuesFrom) e restrição de valor (owl:allValuesFrom) são, respectivamente, equivalentes aos construtores DL de restrição existencial ($\exists R.C$) e restrição de valor ($\forall R.C$). Um exemplo envolvendo ambos os contrutores é apresentado na Figura 9, que representa o axioma "todos os atores premiados estrelaram apenas em filmes e ganharam pelo menos um globo de ouro".

```
<owl:Class rdf:about="http://cinema.edu#AwardWinningActor">
 <rdfs:subClassOf>
  <owl:Class>
   <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
     <owl:onProperty
        rdf:resource="http://cinema.edu#hasWon"/>
     <owl:someValuesFrom
        rdf:resource="http://cinema.edu#GoldenGlobe"/>
    </owl:Restriction>
    <owl:Restriction>
     <owl:onProperty
        rdf:resource="http://cinema.edu#hasFilmed"/>
     <owl:allValuesFrom
        rdf:resource="http://cinema.edu#Movie"/>
    </owl:Restriction>
   </owl:intersectionOf>
  </owl:Class>
 </rdfs:subClassOf>
</owl:Class>
```

Figura 9 – Exemplo de declaração a respeito dos construtores de restrição existencial e restrição de valor em uma ontologia OWL 2 DL na sintaxe RDF/XML

OWL 2 DL também suporta formalismos terminológicos mais expressivos capazes de permitir declarações como "'um ator ou uma atriz estrelou em pelo menos uma série de TV ou um filme", conforme exemplificado na Figura 10, a seguir.

```
<owl:Class>
 <owl:unionOf rdf:parseType="Collection">
  <rdf:Description rdf:about="http://cinema.edu#Actor"/>
  <rdf:Description rdf:about="http://cinema.edu#Actress"/>
 </owl:unionOf>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty
    rdf:resource="http://cinema.edu#hasFilmed"/>
   <owl:someValuesFrom>
    <owl:Class>
     <owl:unionOf rdf:parseType="Collection">
      <rdf:Description rdf:about="http://cinema.edu#Movie"/>
      <rdf:Description rdf:about="http://cinema.edu#TVSeries"/>
     </owl:unionOf>
    </owl:Class>
   </owl:someValuesFrom>
  </owl:Restriction>
</rdfs:subClassOf>
</owl:Class>
```

Figura 10 – Exemplo de declaração a respeito de formalismos terminológicos mais expressivos em uma ontologia OWL 2 DL na sintaxe RDF/XML

Outro construtor OWL 2 DL é conhecido como *owl:equivalentClass*, que representa o construtor DL de equivalência, por exemplo, pode-se definir o nome *AwardedWinningActor* como uma abreviação para a classe que abrange "todos os indivíduos que são atores e também ganharam pelo menos um prêmio", conforme representado na Figura 11.

```
<owl:Class rdf:about="http://cinema.edu#AwardWinningActor">
 <owl:equivalentClass>
  <owl:Class>
   <owl:intersectionOf rdf:parseType="Collection">
    <rdf:Description rdf:about="http://cinema.edu#Actor"/>
    <owl:Restriction>
     <owl:onProperty rdf:resource="http://cinema.edu#hasWon"/>
     <owl:someValuesFrom rdf:resource="http://cinema.edu#Award"/>
    </owl:Restriction>
   </owl:intersectionOf>
  </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

Figura 11 – Exemplo de declaração a respeito do construtor de equivalência em uma ontologia OWL 2 DL na sintaxe RDF/XML

O construtor OWL 2 DL denominado *owl:inverseOf* corresponde ao construtor DL de

propriedade inversa ($^-$), por exemplo, o axioma "todo filme é estrelado por pelo menos um ator ou atriz" pode ser representado como visto na Figura 12.

```
<owl:Class rdf:about="http://cinema.edu#Movie">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty>
    <rdf:Description>
     <owl:inverseOf
        rdf:resource="http://cinema.edu#hasFilmed"/>
    </rdf:Description>
   </owl:onProperty>
   <owl:someValuesFrom>
    <owl:Class>
     <owl:unionOf rdf:parseType="Collection">
      <rdf:Description rdf:about="http://cinema.edu#Actor"/>
      <rdf:Description rdf:about="http://cinema.edu#Actress"/>
     </owl:unionOf>
    </owl:Class>
   </owl:someValuesFrom>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Figura 12 – Exemplo de declaração a respeito do construtor de propriedade inversa em uma ontologia OWL 2 DL na sintaxe RDF/XML

Todos os construtores OWL 2 DL discutidos acima são utilizados em declarações TBox. Com relação à parte ABox de uma ontologia OWL 2 DL, é possível declarar um indivíduo especificando a classe ao qual ele pertence, por exemplo, o indivíduo *Óscar de Melhor Atriz* pode ser declarado como uma instância da classe de Óscars, como representado na Figura 13.

```
<owl:NamedIndividual
    rdf:about="http://cinema.edu#OscarsBestActress">
    <rdf:type
        rdf:resource="http://cinema.edu#Oscar"/>
</owl:NamedIndividual>
```

Figura 13 – Exemplo de declaração de um indivíduo em uma ontologia OWL 2 DL na sintaxe RDF/XML

Uma ontologia OWL 2 DL ainda possibilita a declararação de um indivíduo e seus relacionamentos com outros indivíduos, por exemplo, "Julia Roberts é uma atriz e ganhou o Óscar de Melhor Atriz" pode ser representado conforme visto na Figura 14, a seguir.

Na Seção 2.2, a seguir, são apresentadas informações quanto à raciocínio automático sobre ontologias.

```
<owl:NamedIndividual
    rdf:about="http://cinema.edu#JuliaRoberts">
    <rdf:type
        rdf:resource="http://cinema.edu#Actress"/>
    <hasWon
        rdf:resource="http://cinema.edu#OscarsBestActress"/>
</owl:NamedIndividual>
```

Figura 14 – Exemplo de declaração de um indivíduo e seus relacionamentos em uma ontologia OWL 2 DL na sintaxe RDF/XML

2.2 Raciocínio Automático sobre Ontologias

Um sistema de raciocínio automático objetiva a descoberta de novos fatos a partir de informações definidas nas bases de conhecimento. No geral, o objetivo é avaliar uma ontologia de acordo com sua consistência e classificação de seus conceitos (e.g. classes, relacionamentos e indivíduos) e por meio de algum conjunto de regras de inferência[8]. A regra de subsunção, por exemplo, é a que permite conclusões a respeito da hierarquia de classes em uma ontologia, assim, uma subclasse herda todas as características de suas superclasses.

Os sistemas de raciocínio podem ser categorizados conforme as linguagens de representação de conhecimento utilizadas, como, por exemplo, Motores de Inferência OWL 2 DL e Provadores Automáticos de Teoremas em FOL. Apesar das diferentes categorias, esses tipos de raciocinadores implementam funções que possibilitam consultas sobre os conhecimentos de uma ontologia, checagem de consistência, classificação e hierarquia de conceitos, entre outras funcionalidades.

Nas Subções 2.2.1 e 2.2.2, a seguir, são apresentados dois motores de inferência OWL 2 DL, respectivamente, *Pellet* e *RACCOON*. Na Subseção 2.2.3 é apresentado o *E Prover*, um sistema ATP para FOL.

2.2.1 Pellet

Pellet (PARSIA; SIRIN, 2004) é um motor de inferência desenvolvido em Java e de código aberto, ele implementa uma versão do método de Tableaux[9] para realizar inferências sobre ontologias OWL 2 DL. Algumas de suas principais funcionalidades são:

- Checagem de consistência: função que busca encontrar e demonstrar fatos contraditórios avaliando a consistência de uma ontologia.

[8] Em lógica, regras de inferência podem ser entendidas como regras de transformação sintáticas e semânticas que podem ser usadas para inferir uma conclusão a partir de uma premissa

[9] Na teoria da prova, o tableau semântico é um sistema de dedução e um procedimento de prova para fórmulas da lógica de primeira ordem

- Classificação: função capaz de determinar a hierarquia de conceitos (e.g. classes e relacionamentos) da ontologia.
- Consultas (Conjecturas): função que permite verificar se um determinado axioma (conjectura) é inferido pelos conhecimentos da ontologia.
- Realização: função para identificar as classes de cada indivíduo na base de conhecimento.

O *Pellet* compreende uma ampla variedade de construtores DL, inclusive os que foram discutidos na Seção 2.1.1.3, por conta disso, é um raciocinador bastante utilizado na área da Engenharia de Ontologias.

2.2.2 RACCOON

RACCOON (MALHEIROS; FREITAS, 2017) (Reasoner based on the Connection Calculus Over ONtologies, em tradução livre, Raciocinador baseado no Cálculo de Conexões Sobre Ontologias) é um motor de inferência para ontologias OWL 2 DL. Ele se baseia no método de conexões proposto por Freitas e Otten (2016), ALC θ-CM, como foi chamado, é uma adaptação do método de conexões para FOL criado por W. Bibel nos anos 70.

O *RACCOON* foi desenvolvido na linguagem de programação *C++* e é capaz de realizar raciocínio automático para verificar a consistência de ontologias OWL 2 DL com expressividade ALC (*Attributive Concept Language with Complements*, em tradução livre Linguagem Conceitual Atributiva com Complementos). Sua principal característica é demandar o uso de pouca memória computacional se comparado a outros motores de inferência, todavia, ainda não implementa outros serviços de inferência como, por exemplo, consultas e classificação de hierarquia de classes.

2.2.3 E Prover

E Prover[10] é um provador automático de teoremas (sistema ATP) para FOL com operador de igualdade, de código aberto e desenvolvido na linguagem de programação *C*. Ele foi criado como parte do projeto E-SETHEO da Universidade Técnica de Munique [11], sendo publicado pela primeira vez no ano de 1998 e melhorado até os dias atuais (SCHULZ, 2013).

O *E Prover* permite a especificação de um problema, geralmente consistindo de um conjunto de cláusulas em FOL e uma conjectura[12]. Seu objetivo é encontrar uma prova

[10] https://wwwlehre.dhbw-stuttgart.de/ sschulz/E/E.html
[11] http://www.in.tum.de/startseite/
[12] Conjectura pode ser entendida como o ato de inferir ou deduzir, por meio de hipóteses e suposições, que algo é provável

formal para a conjectura por meio de raciocínio sobre os axiomas na base de conhecimento. Além disso, se a conjectura envolve o quantificador existencial (e.g. "Existe algum indivíduo X que é uma instância da classe Humano"), a última versão do sistema é capaz de apresentar várias respostas com possíveis valores (instâncias) para as variáveis existenciais (e.g. X).

O *E Prover* tem participado constantemente de competições da CASC[13] (*CADE ATP System Competition*, em tradução livre Competição de Sistemas ATP da CADE), com o objetivo de comparar a sua eficiência em relação a outros sistemas ATP. Na edição de 2018, o sistema alcançou o terceiro lugar na categoria denominada FOF (First Order Formula, em tradução livre Fórmula de Primeira Ordem).

2.3 Ontologia de Topo SUMO

A *Suggested Upper Merged Ontology* (SUMO, em tradução livre Ontologia Mesclada de Topo Sugerida) é uma ontologia de topo desenvolvida e apresentada pelo SUO WG (*Standard Upper Ontology Working Group*, em tradução livre Grupo de Trabalho para a Ontologia de Topo Padrão), que objetivou a criação da *Standard Upper Ontology* (SUO, em tradução livre Ontologia de Topo Padrão) (NILES; PEASE, 2001a; NILES; PEASE, 2001b).

A SUMO foi construída através da linguagem de representação de conhecimento SUO-KIF (Vide Seção 2.1.1.1) e possui a definição de cerca de 25 mil termos e 80 mil axiomas, com isso, tornou-se uma das maiores ontologias gratuitas que proporcionam o reuso de informações semanticamente definidas na *Web*. Em contrapartida, a Engenharia de Ontologias ainda não dispõe de sistemas capazes de raciocinar sobre uma base de conhecimento representada em SUO-KIF.

Para realizar raciocínio automático sobre uma ontologia em SUO-KIF, faz-se necessário a tradução da base de conhecimento para a linguagem FOL. Em razão disso, foi desenvolvida a ontologia *Adimen-SUMO* (ÁLVEZ; LUCIO, 2012), representada em FOL através de reengenharia de aproximadamente 88% da ontologia SUMO. Álvez, Lucio e Rigau (2015) conduziram experimentos mostrando que a *Adimen-SUMOv2.4* desempenhou melhores resultados se comparada a versão em SUO-KIF, contribuíndo assim com ferramentas que desejam usar informações inferidas pela SUMO.

[13] http://tptp.cs.miami.edu/ tptp/CASC/

2.4 Busca em Largura

Um algoritmo de busca fraciona um problema em dois conjuntos, respectivamente, espaço de estados e ações. Um estado é uma abstração sobre alguma entidade do mundo real, já uma ação é uma atividade executada para levar a busca de um estado origem para outro estado destino. Para exemplificar, pode-se considerar o problema de encontrar a menor rota entre duas cidades da Romênia, de acordo com a Figura 15, a seguir. O espaço de estados é composto pelas cidades, e uma ação seria sair de uma cidade com destino a outra por meio de uma rodovia. Segundo Russell e Norvig (2009) um problema pode ser definido formalmente por cinco componentes:

- **Estado inicial**: a definição do estado por onde a busca começa. Um estado inicial do exemplo da Romênia poderia ser descrito como *Em(Arad)*.
- **Ações**: uma descrição das ações possíveis que estão disponíveis para a busca. Dado um estado *s*, *AÇÕES(s)* devolve o conjunto de ações que podem ser executadas em *s*. Dizemos que cada uma dessas ações é aplicável em *s*. Por exemplo, através do estado *Em(Arad)*, as ações aplicáveis são: *Ir(Sibiu)*, *Ir(Timisoara)*, e *Ir(Zerind)*.
- **Modelo de transição**: uma descrição do que cada ação realiza, *RESULTADO(s, a)* devolve o estado - também chamado sucessor - que resulta da ação *a* aplicada ao estado *s*. Usa-se também o termo sucessor como referência a um estado acessível a partir de uma única ação sobre outro estado. Por exemplo, temos *RESULTADO(Em(Arad), Ir(Zerind)) = Em(Zerinde)*. O estado inicial, as ações, e o modelo de transição definem de forma implícita o espaço de estados - o conjunto de todos os estados acessíveis a partir do estado inicial.
- **Teste de objetivo**: uma função que determina se um certo estado é um estado objetivo. Atingir um estado objetivo significa que uma solução para o problema foi encontrada pela busca. Por exemplo, o objetivo da busca do problema da Romênia é o conjunto unitário *Em(Bucareste)*.
- **Custo de caminho**: função que atribui um custo numérico a cada caminho da busca, isto é, uma sequência de estados conectados por uma sequência de ações. Isso resulta a soma dos custos das ações aplicadas individualmente ao longo do caminho. No exemplo da Romênia, o custo da ação *Ir* do estado *Arad* para *Zerind* é igual a 75 (setenta e cinco).

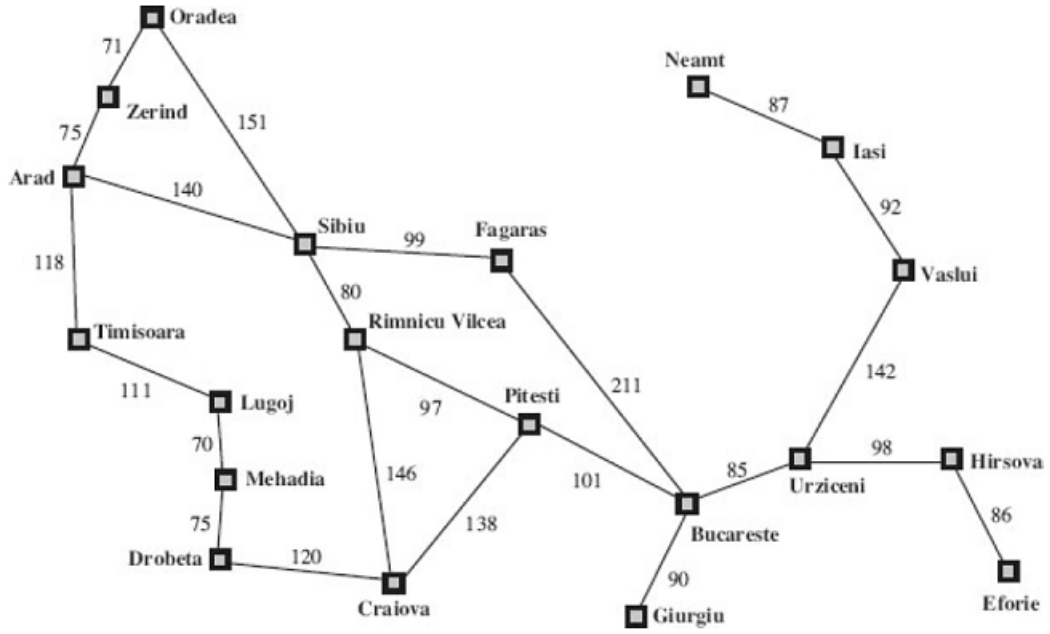

Figura 15 – Mapa rodoviário simplificado de parte da Romênia

Fonte: Russell e Norvig (2009)

O primeiro passo do processo de busca é criar a borda, uma lista de caminhos ainda inexplorados. A borda é inicializada com um caminho com custo zero referente ao estado inicial. O segundo passo é obter e remover o caminho, denominado pai, no início da borda. Caso o estado no final desse caminho esteja no conjunto objetivo, a busca pode ser encerrada com uma solução válida - isto é, uma sequência de ações aplicadas. Do contrário o terceiro passo é conseguir as ações aplicáveis ao estado final de seu pai. Cada ação gera um novo caminho - denominado filho - que é adicionado na borda, sendo seu custo igual a soma do custo da ação e do custo de seu pai. Por fim, o segundo passo é repetido, exceto na hipótese de falha na busca, isto é, a borda estar vazia.

O algoritmo Busca em Largura (BL) segue o processo descrito acima, sua diferença em relação aos outros algoritmos de busca refere-se à organização e ordenação da borda. Ela organiza a borda de tal forma que os novos caminhos sejam adicionados no final, assim prioriza expandir os estados ao redor do estado inicial. O Algoritmo 1, a seguir, apresenta um modelo de sua implementação.

Algorithm 1 Algoritmo Busca em Largura em um grafo

Entrada:

- *caminho-de-inicio* = (EstadoInicial, 0, , Nula, Nulo) // onde Caminho = (Estado, Custo-Caminho, Ação, Pai)

Saída:

- *solução* // onde solução retorna falha ou um caminho que a partir do estado inicial alcança um estado objetivo

```
1: função BUSCA-EM-LARGURA
2:     borda ← { Caminho }                    ▷ uma lista de caminhos ainda inexplorados
3:     borda.Adicionar(caminho-de-inicio)
4:     explorado ← { Estado }                 ▷ uma lista de estados já explorados pela busca
5:     enquanto NãoVazia?(borda) faça
6:         pai ← POP(borda)                   ▷ pega e remove o elemento no início da borda
7:         se TesteObjetivo(pai.Estado) então devolve pai
8:         senão
9:             explorado.Adicionar(pai.Estado)
10:            para ação em AÇÔES(pai.Estado) faça
11:                estado-destino ← RESULTADO(pai.Estado, ação)
12:                se estado-destino não está em explorado então
13:                    filho ← (estado-destino, ação.Custo+pai.Custo-Caminho, ação, pai)
14:                    se estado-destino não está na borda então
15:                        borda.AdicionarNoFinal(filho)
16:                    fim se
17:                fim se
18:            fim para
19:        fim se
20:    fim enquanto
       devolve falha
21: fim função
```

Para exemplificar a busca em largura, pode-se considerar o mesmo problema da Romênia de forma simplificada, conforme a Figura 16, a seguir. A busca objetiva encontrar uma rota entre as cidades Sibiu e Bucareste. Primeiro a borda é criada com um único caminho ({ Sibiu }) com custo zero referente ao estado inicial. Primeiro o algoritmo seleciona *Sibiu*, no início da borda, e gera dois filhos através das ações *Ir(Fagaras) e Ir(Rimnincu Vilcea)*. Neste hora a borda contém, respectivamente: *{ Sibiu → Fagaras }* com custo 99; e *{ Sibiu → Rimnincu Vilcea }* com custo 80. Assim o caminho a ser expandido é *Fagaras*, gerando *{ Sibiu → Fagaras → Bucareste }* com custo de $99 + 211 = 310$. Em seguida o caminho explorado é *Rimnincu Vilcea*, gerando *{ Sibiu → Rimnincu Vilcea → Pitesti }* com custo $80 + 97 = 177$. A partir de agora a borda possui *{ Sibiu → Fagaras → Bucareste }* e *{ Sibiu → Rimnincu Vilcea → Pitesti }*, nessa mesma ordem. Portanto o estado final do caminho a ser selecionado refere-se à *Bucareste*, levando o algoritmo a encerrar sua execução com uma solução válida.

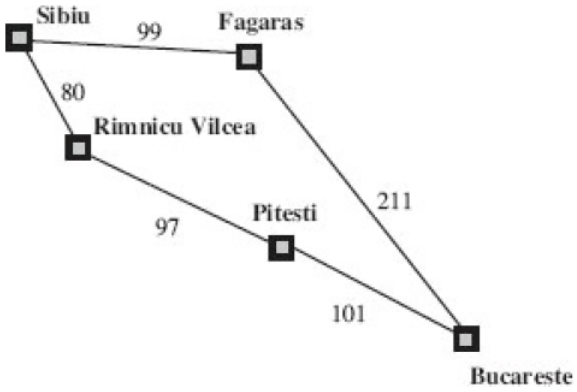

Figura 16 – Parte do espaço de estados da Romênia, selecionada para ilustrar a Busca em Largura

Fonte: Russell e Norvig (2009)

3 COREACQ

O principal propósito desta dissertação consistiu em definir o *CoreACQ*, um *framework* para empregar raciocínio sobre a ontologia SUMO com o intuito de realizar validação de CQs de forma automática. Outro objetivo foi desenvolver procedimentos capazes de agilizar o processo de raciocínio em um sistema ATP e, assim, minimizar o tempo de resposta das inferências. Um mecanismo de *cache* também é proposto para armazenar validações a fim de evitar reexecução de inferências já concretizadas. Além disso, o dicionário *WordNet*, um dos componentes da arquitetura, é apresentado como uma solução para problemas relacionados a identificação de conceitos desconhecidos.

A principal motivação para um *framework* de validação de CQs baseado na SUMO foi beneficiar ferramentas interessadas em autonomia para evoluir ontologias de domínio automaticamente. A arquitetura do *CoreACQ* foi concebida com o princípio de torná-lo o mais reusável possível, sendo toda baseada em componentes, visando sobretudo permitir o seu uso por ferramentas da engenharia de ontologias.

Na perspectiva acima, *CoreACQ*, deve providenciar componentes apropriados acoplados em sua arquitetura, como por exemplo sistemas ATP. Na sequência é apresentada a arquitetura do *framework*, desenvolvido principalmente para manipular bases de conhecimento da SUMO em FOL e realizar raciocínio automático a partir de seus axiomas. Seus componentes, tecnologias utilizadas, bem como, exemplos de funcionamento são detalhados nas seções a seguir.

3.1 Arquitetura

Com o objetivo de seguir padrões e boas práticas de desenvolvimento, é viável utilizar artefatos produzidos pela Engenharia de *Software* baseada em componentes (CBSE, *component-based software engineering*). Pressman (2002) define a CBSE como um processo que enfatiza o projeto e a construção de sistemas baseados em computador usando componentes de *software* reutilizáveis. A Engenharia de Componentes é uma subárea da Engenharia de *Software* focada na decomposição de um sistema - ou subsistema, e.g. *framework* computacional - em componentes funcionais e lógicos com interfaces bem definidas.

As interfaces definidas pela Engenharia de Componentes são usadas para comunicação entre os próprios componentes em um alto nível de abstração, realizada sobretudo por troca de mensagens. Para exemplificar, um componente envia uma mensagem para outro especificando algo a ser feito, o segundo então processa a solicitação do primeiro que, no final, recebe o resultado da operação em questão. Assim um componente pode tirar proveito das funcionalidades de outros, precisando entender a entrada e a saída de dados,

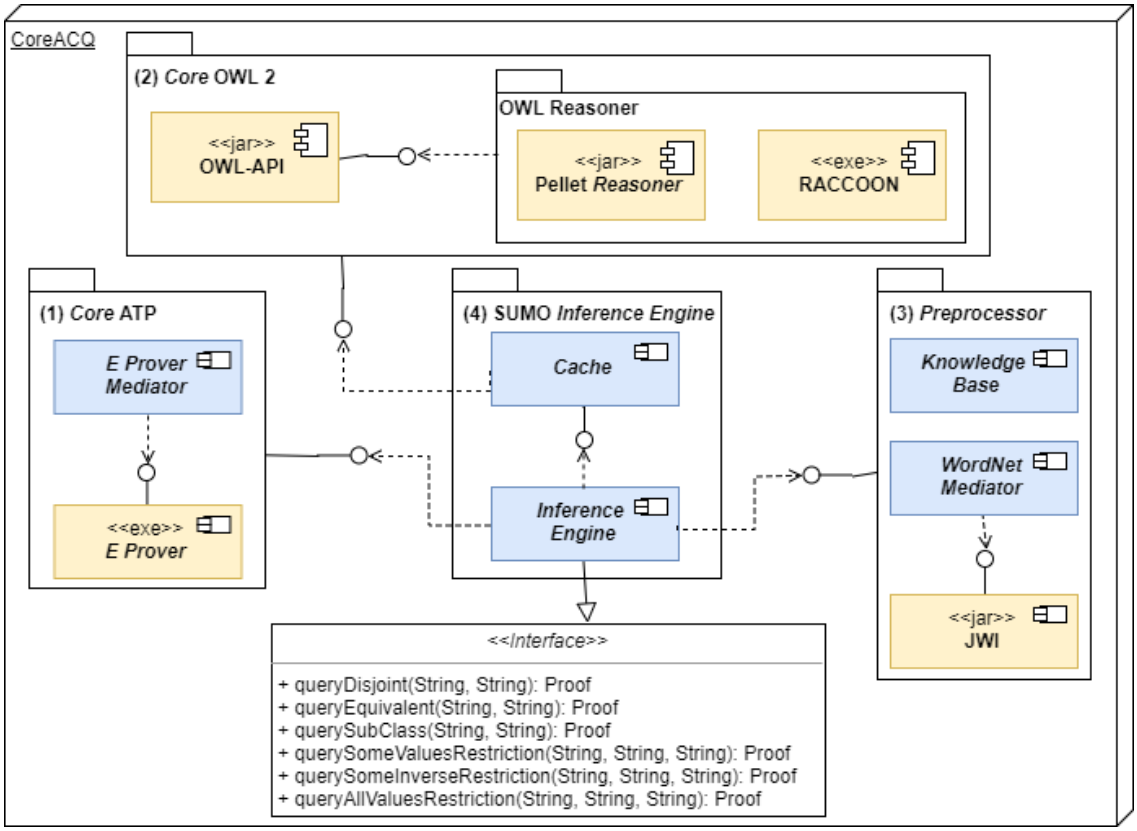

Figura 17 – Diagrama de componentes

mas sem necessitar compreender detalhes específicos de implementação. Além do mais, os componentes podem ser separados em conjuntos denominados pacotes ou módulos, tendo em conta seus propósitos dentro de uma arquitetura.

A adoção da abordagem CBSE tende a facilitar a construção, manutenção, interoperabilidade, isolamento e distribuição dos componentes em módulos essenciais do *CoreACQ*. Portanto sua arquitetura foi dividida em quatro módulos com subsistemas e funções bem definidos.

A visão geral da arquitetura do *CoreACQ* é exibida através do diagrama de componentes na Figura 17. Os módulos são: (1) ***Core ATP***, que possui um componente com funções específicas e é responsável pela comunicação com sistemas ATP; (2) ***Core OWL***, contendo três componentes particulares é o componente responsável por dispor recursos para manipulação de ontologias e motores de inferência OWL 2 DL; (3) ***Preprocessor*** (Pré-processador), que contém dois componentes próprios e é responsável por organizar em memória computacional a base de conhecimento da SUMO em FOL e, também, exercer comunicação com o dicionário WordNet; e (4) ***SUMO Inference Engine*** (Motor de Inferência SUMO), abrangendo seus dois componentes.

O diagrama de componentes é utilizado para descrever a organização dos elementos em um sistema, quais componentes pertencem a cada módulo (ou pacote) do diagrama e, ainda, as dependências entre eles. Nesses termos, uma dependência é denotada em forma

de uma seta pontilhada, indicando que o módulo do qual ela parte necessita de funcionalidades do módulo que a recebe. Por exemplo, observa-se na arquitetura apresentada (Vide Figura 17) que o componente (4) ***SUMO Inference Engine*** que depende de atividades exercidas pelos outros três módulos e é o responsável por manipular CQs e fazer uso dos outros módulos para executar raciocínio automático e armazenar validações realizadas.

Nas subseções seguintes são descritos em detalhes os elementos apresentados na Figura 17, principalmente a respeito dos módulos principais mencionados anteriormente.

3.1.1 Core ATP

Este módulo engloba os componentes responsáveis pela comunicação com sistemas ATP, como visto na Figura 18, implementando na versão atual componentes relacionados ao *E Prover ATP* (Vide Seção 3.2.4).

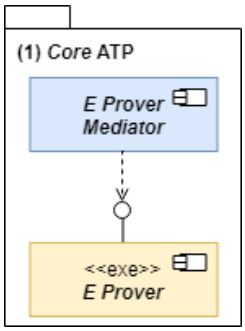

Figura 18 – Diagrama simplificado de componentes do módulo *Core ATP*

Do ponto de vista deste trabalho, raciocínio automático por ATP é realizado por demanda, isto é, por consulta. Cada consulta busca averiguar se um axioma (conjectura) é uma consequência lógica da base de conhecimento, ou seja, se a conjectura é deduzida por intermédio dos axiomas da base (Vide Figura 19). Em relação ao *E Prover* é oferecida uma interface de comunicação interativa, que recebe uma consulta, ativa o raciocínio, fornece a respectiva resposta e aguarda uma nova consulta. Quando inicializada expõe uma mensagem para indicar que está pronta para executar seus serviços, conforme exibido na linha 4 da Figura 19.

É apresentado na Figura 19 um exemplo do processo de comunicação com o *E Prover*. No exemplo, é possível observar a declaração de dois axiomas e uma consulta a ser realizada. O primeiro (*predefinitionsA12*), linha 5 a 7, determina que uma classe (conjunto) possui todos os indivíduos (elementos) de suas subclasses (subconjuntos), assim como ocorre a inclusão em teoria dos conjuntos. O outro (*merge195A2983*), linha 9 a 11, destaca um relacionamento de subsunção entre mulheres (*c___Woman*) e humanos (*c___Human*), isto é, toda mulher é um humano. Já a consulta (*query1*), linha 13 a 15, visa verificar se qualquer indivíduo cujo gênero é mulher também possui como característica ser humano.

```
# == WCT:    0s, Solved:    0/   0    ==
# =============== Batch done ===========

# Enter job, 'help' or 'quit', followed by 'go.' on a line of its own:
fof(predefinitionsA12, axiom,
  (![X,Y,Z]: (( $p__instance(X,Y) & $p__subclass(Y,Z) ) => $p__instance(X,Z) ))
). // Primeiro axioma

fof( merge195A2983, axiom,
  $p__subclass(c__Woman, c__Human)
). // Segundo axioma

fof( query1, conjecture,
  (![X]: (( $p__instance(X, c__Woman) ) => ( $p__instance(X, c__Human) ) ))
). // Consulta

go.
```

Figura 19 – Exemplo de inicialização do *E Prover* para raciocínio por consulta

O *E Prover* ao realizar o raciocínio automático tenta encontrar um grupo de axiomas cuja consequência lógica seja *query1*, do mesmo modo que acontece com *predefinitionsA12* e *merge195A2983*. Na Figura 20 é exibida a sua resposta para a consulta, mostrando o texto '*Proof found*' (em tradução livre, 'Prova Encontrada') e informações sobre os axiomas considerados no procedimento, como visto nas linhas 19 a 23.

```
# Proof found!
# SZS status Theorem
# SZS output start CNFRefutation
 fof(predefinitionsA12, axiom, ... ).
 fof( merge195A2983, axiom, ...).
 ************
['proof']).
# SZS output end CNFRefutation
# Training examples: 0 positive, 0 negative

# ------------------------------------------------
# User time                : 0.008 s
# System time              : 0.000 s
# Total time               : 0.008 s
# Maximum resident set size: 4136 pages
Terminated
Terminated

# Processing finished for unnamed_job
```

Figura 20 – Exemplo de resposta do *E Prover* para raciocínio por consulta

Assim sendo, foi desenvolvido o componente ***E Prover Mediator*** com a finalidade de intermediar a comunicação entre *CoreACQ* e o *E Prover*. Esse módulo torna-se essencial devido a viabilizar inferências sobre bases de conhecimento da SUMO em FOL.

3.1.2 Core OWL

Este módulo possui recursos necessários para manipulação de ontologias e inferências sobre OWL 2 DL, como visto na Figura 21, a seguir, utilizando principalmente a biblioteca OWL API (Vide Seção 3.2.1). Seu objetivo é disponibilizar funções especializadas em

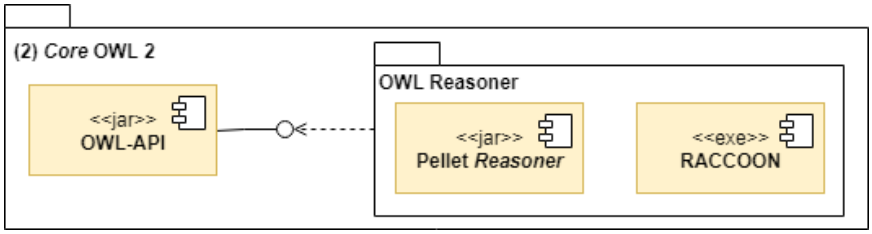

Figura 21 – Diagrama simplificado de componentes do módulo *Core OWL*

```
2 <rdf:RDF xmlns="http://www.example.com#"
3  xml:base="http://www.example.com"
4  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5  xmlns:owl="http://www.w3.org/2002/07/owl#"
6  xmlns:xml="http://www.w3.org/XML/1998/namespace"
7  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
8  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
9 <owl:Ontology rdf:about="http://www.example.com"/>
10 <owl:Class rdf:about="http://www.example.com#Mammal"/>
11 <owl:Class rdf:about="http://www.example.com#Human">
12  <rdfs:subClassOf rdf:resource="http://www.example.com#Mammal"/>
13 </owl:Class>
14 <owl:Class rdf:about="http://www.example.com#Woman">
15  <rdfs:subClassOf rdf:resource="http://www.example.com#Human"/>
16 </owl:Class>
17 </rdf:RDF>
```

Figura 22 – Exemplo de uma ontologia na linguagem OWL 2

simplificar a criação e manutenção de uma ontologia em aplicações Java[1]. Na Figura 22 é apresentado um exemplo envolvendo declarações de três classes na linguagem OWL 2 DL. A primeira diz respeito a classe Mamífero (*Mammal*), conforme a linha 10. A segunda refere-se a classe Humano (*Human*) definida como uma subclasse de Mamífero (*Human* $\sqsubseteq$ *Mammal*), como visto nas linhas 11 a 13. A última define a classe Mulher (*Woman*) como sendo uma subclasse de Humano (*Woman* $\sqsubseteq$ *Human*), de acordo com as linhas 14 a 16.

Em se tratando de raciocínio sobre ontologias OWL 2 DL, vem a ser importante a adoção de um motor de inferência como *Pellet* ou *RACCOON* (Vide Seção 3.2.2). Por intermédio deles é possível deduzir novos conhecimentos através do conteúdo em uma ontologia. No mesmo exemplo da Figura 22, esses sitemas (raciocinadores) são capazes de processar uma consulta DL para inferir que todo indivíduo mulher tem como atributo ser mamífero (*Woman* $\sqsubseteq$ *Mammal*).

Dessa maneira, este módulo oferece funcionalidades para manipulação de ontologias OWL 2 DL por meio da OWL API e utiliza as máquinas de inferência *Pellet* ou *RACCOON* para realizar raciocínio automático.

[1] Java é uma linguagem de programação orientada a objetos desenvolvida na década de 90 por uma equipe de programadores chefiada por James Gosling, na empresa *Sun Microsystems*

3.1.3 Preprocessor

Este módulo contém os componentes para o pré-processamento de uma CQ a fim de selecionar premissas e identificar conceitos desconhecidos para o processo de raciocínio, conforme visto na Figura 23, a seguir. Ele organiza os conhecimentos da ontologia SUMO em memória computacional, usando uma estrutura em formato de um grafo para facilitar o acesso aos conceitos e axiomas em momentos futuros.

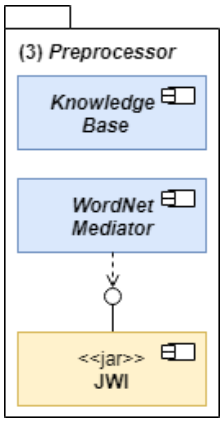

Figura 23 – Diagrama simplificado de componentes do módulo *Preprocessor*

Na situação em que a SUMO não compreende o significado de um conceito, nada pode ser concluído a seu respeito. No entanto, a existência de um sinônimo conhecido proporciona que seu significado seja atribuído ao conceito desconhecido. Além disso, a presença de um hiperônimo[2] também oportuniza que suas definições sejam herdadas por outros conceitos. Um exemplo disso acontece com o conceito *YoungWoman* e seu sinônimo *Girl*, relacionados por intervenção de dicionários existentes. A SUMO define *Girl* como sendo um tipo de *Woman* (e.g. $Girl \sqsubseteq Woman$), tornando possível conclusões como *YoungWoman* é um tipo de *Woman* (e.g. $YoungWoman \sqsubseteq Woman$). Por isso a tecnologia JWI (Vide Seção 3.2.3) é aplicada para propiciar um canal de acesso ao dicionário *WordNet*.

Por outro lado, a seleção de premissas visa reduzir o tamanho da base de conhecimento conforme uma representação em grafo e um algoritmo de busca. Na Figura 24 é apresentado um exemplo de conceitos (nós) ligados por meio de seus axiomas (arestas). Uma extensão do algoritmo de Busca em Largura (BL) (Vide Seção 2.4 do Capítulo 2) é implementada para selecionar todas as soluções possíveis, isto é, caminhos cujos nós inicial e final fazem referência aos conceitos (círculo duplo) de uma CQ ou suas superclasses (tracejados). Nessa extensão o custo de caminho diz respeito à quantidade de arestas que o compõem, por exemplo $\{Ear \rightarrow 2 \rightarrow Head \rightarrow 4 \rightarrow BodyPart\}$ envolve duas arestas, ou seja, tem custo 2. Inicialmente é criado um caminho de custo zero para cada conceito e cada superclasse relacionados à CQ de entrada.

[2] Hiperônimos são palavras de sentido mais genérico, cujos significados são mais abrangentes do que outras palavras do mesmo campo semântico

O algoritmo de busca implementado escolhe iterativamente um caminho - necessariamente aquele adicionado primeiro dentre os caminhos possíveis - e expande-o através do nó final e suas respectivas arestas. Ao contrário da abordagem original, a busca adotada grava todos as soluções descobertas e encerra sua execução ao atingir um custo de caminho máximo definido previamente.

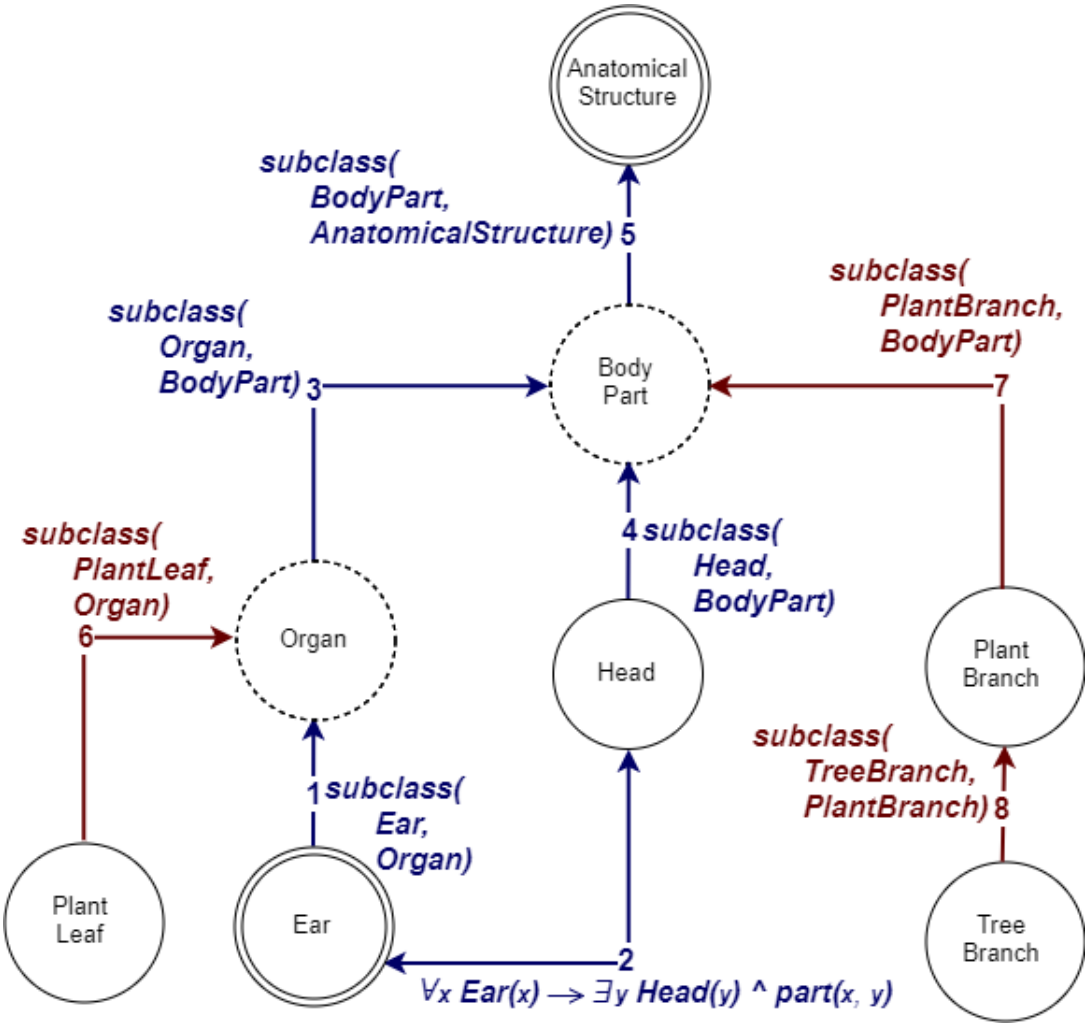

Figura 24 – Exemplo de Busca em Largura para o procedimento de seleção de premissas do *CoreACQ*

O exemplo da Figura 24 envolve uma consulta que tem em vista provar se todo ouvido faz parte de alguma estrutura anatômica ($Ear \sqsubseteq \exists part.AnatomicalStructure$). A busca primeiro expande, respectivamente: *Ear* pelas arestas 1 e 2; *AnatomicalStructure*, sem arestas; *Organ* pela aresta 3; e *BodyPart* através da aresta 5. Em seguida, o nó *Head* é expandido pela aresta 4. Neste ponto não existem novos caminhos para serem expandidos, portanto a busca é encerrada. O resultado do algoritmo é um subconjunto da base de conhecimento que engloba os axiomas 1, 2, 3, 4 e 5, ou seja, aqueles pertencentes às soluções (caminhos) encontradas.

A seleção de premissas tem como objetivo reunir axiomas sem relevância em relação

a uma CQ de entrada, evitando que esses sejam transmitidos ao sistema ATP (e.g. *E Prover*). Com isso espera-se uma otimização no processo de raciocínio, de preferência uma redução no tempo de resposta (validação). Contudo tal método baseia-se completamente em um algoritmo procedimental que pode causar impactos significativos no que se refere à Completude do *CoreACQ* - assunto a ser discutido com mais detalhes na Seção 4.2 do Capítulo 4.

Sendo assim, este módulo é formado por dois componentes, são eles: ***Knowledge Base***, responsável por organizar em memória computacional uma base de conhecimento e, ainda, pelo método de seleção de premissas; e ***WordNet Mediator***, desenvolvido para intermediar a comunicação com o componente *WordNet*.

3.1.4 SUMO Inference Engine

Este é o módulo central da arquitetura do *CoreACQ*, sobretudo por disponibilizar uma interface de comunicação para ferramentas de terceiros beneficiarem-se das funcionalidades implementadas, conforme visto na Figura 25, a seguir. Seu propósito principal é manipular CQs para validação em sistemas ATP (e.g. *E Prover*) utilizando conhecimentos da SUMO em FOL.

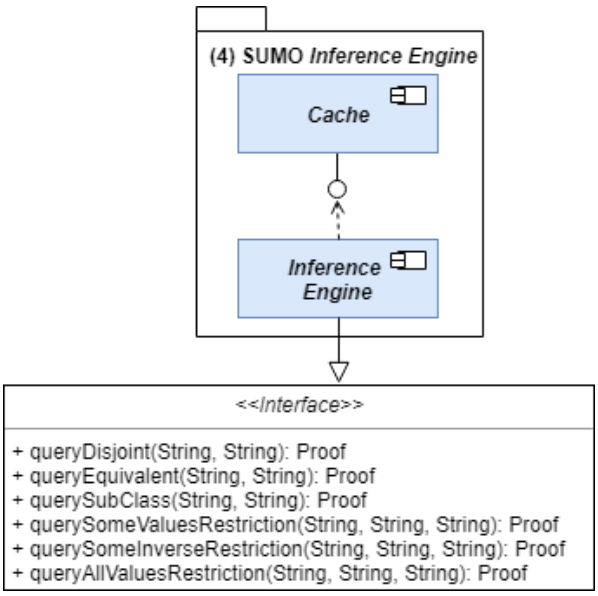

Figura 25 – Diagrama simplificado de componentes do módulo *SUMO Inference Engine*

Considerando-se a dedução em FOL por sistema ATP é importante planejar um meio de evitar repetição de atividades, isto é, validar muitas vezes uma mesma CQ. Principalmente pelo fato de encontrar-se certas inferências que requerem um longo período de tempo, por exemplo em torno de dez (10) minutos, para serem finalizadas. À vista disso, vem a ser relevante o emprego de uma ontologia OWL 2 DL para proporcionar um procedimento de *cache*.

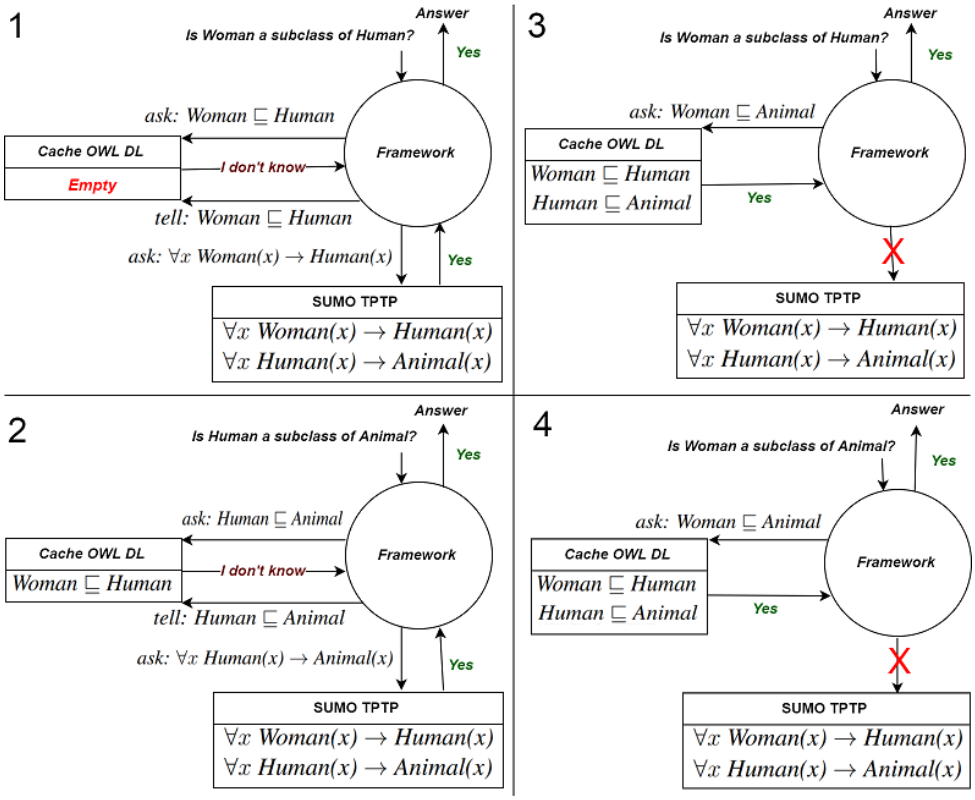

Figura 26 – Exemplo de funcionamento das validações de CQs sobre a *Cache* e a SUMO

É apresentado na Figura 26 o fluxo da interação entre o *CoreATP*, a SUMO e a *Cache* no processamento de validação de CQs. Por meio de uma *interface*, a CQ 1 (*Is Woman a subclass of Human?, em tradução livre Mulher é subclasse de Humano?*) é fornecida como entrada e imediatamente representada em DL ($Woman \sqsubseteq Human$). Em seguida, mediante uma operação *ask* (perguntar), ela é enviada à *Cache* que está vazia e, portanto, não é capaz de deduzir a pergunta. Neste caso, ela é então representada em FOL ($\forall x$ *Woman(x)* $\rightarrow$ *Human(x)*) e transmitida ao sistema ATP, que por sua vez faz uso da SUMO com conhecimento suficiente para realizar a dedução. Logo após a confirmação (*Yes*), a CQ 1 é armazenada na *Cache* por operação *tell* (falar) e a resposta é fornecida como saída. Da mesma maneira, a CQ 2 (*Is Human a subclass of Animal?*, em tradução livre *Humano é uma subclasse de Animal?*) também é respondida através da SUMO e armazenada na *Cache*.

Por outro lado a CQ 3 - idêntica à CQ 1 - é validada apenas por intermédio da representação DL enviada à *Cache*. A razão pela qual uma ontologia é adotada nesse procedimento pode ser observada na validação da CQ 4 (*Is Woman a subclass of Animal?*, em tradução livre *Mulher é uma subclasse de Animal?*), inferida através da cache e seus dois axiomas: (I) toda mulher é um humano; e (II) todo humano é um animal. Essa última dedução torna-se realizada devido a existir motores de inferência OWL 2 DL capazes de raciocinar sobre (I) e (II) para concluir que toda mulher é um animal. Assim sendo, as duas últimas perguntas (3 e 4) foram confirmadas sem a intervenção de um ATP.

Assim, o primeiro componente (***Cache***) deste módulo foi implementado com o objetivo de controlar a ontologia *Cache*, basicamente fazendo uso dos recursos dispostos pelo módulo *Core OWL* (Vide Seção 3.1.2). Ele fornece funções para armazenar e realizar validações e, também, empregar inferências para tentar antecipar a dedução de CQs que ainda não foram processadas (Vide Figura 26).

No que se refere ao segundo componente (***Inference Engine***), ele é responsável por prover uma *interface* pública para dar acesso aos métodos de validação de CQs. Para tal fim, dispõe das funcionalidades oferecidas pelos componentes dos seguintes módulos: ***Core ATP*** (Vide Seção 3.1.1), usado para executar raciocínio automático por sistema ATP; e ***Preprocessor*** (Vide Seção 3.1.3), utilizado para realizar seleção de premissas e identificar conceitos pela *WordNet*, ademais, comunica-se com o primeiro componente deste módulo para acessar a ontologia *Cache*, assim, é pertinente ressaltar que apenas o uso de um sistema ATP é exigido pelo *CoreACQ*, ficando as outras funções opcionais - isto é, procedimento de *cache*, seleção de premissas e consulta à *WordNet*.

Uma ferramenta de terceiros - isto é, um sistema que não foi desenvolvido nesta dissertação - deve comunicar-se com o componente ***Inference Engine*** para fazer uso do *CoreACQ*. Sua *interface* pública compreende seis (6) métodos para manipular diferentes tipos de CQ e realizar validações. Isto posto, a seguir são detalhadas as características e responsabilidades de cada método em questão:

- ***querySubClass***. Este método recebe dois conceitos (e.g. *Woman* e *Human*) ao ser executado. Sua função é comprovar se o primeiro conceito é um tipo específico (subclasse) do segundo (e.g. $Woman \sqsubseteq Human$).

- ***queryDisjoint***. Este método recebe como entrada dois conceitos (e.g. *Woman* e *Man*). Sua finalidade é averiguar se os conceitos recebidos são disjuntos (e.g. $Woman \sqcap Man \sqsubseteq \bot$).

- ***queryEquivalent***. Este método recebe dois conceitos (e.g. *YoungWoman* e *Girl*) ao ser iniciado. Seu objetivo é determinar se os conceitos de entrada são equivalentes (e.g. $YoungWoman \equiv Girl$).

- ***querySomeValuesRestriction***. Este método recebe dois conceitos (e.g. *Father* e *Child*) e uma relação (e.g. *fatherOf*) como entrada. Sua atribuição é provar se todo indivíduo referente ao primeiro conceito possui um certo relacionamento com algum indivíduo pertencente ao segundo conceito, por exemplo, se todo pai tem ao menos um filho (e.g. $Father \sqsubseteq \exists fatherOf.Child$).

- ***querySomeInverseRestriction***. Este método recebe dois conceitos (e.g. *Father* e *Child*) e uma relação (e.g. *fatherOf*) ao ser executado. Sua inteção é demonstrar se todo indivíduo referente ao primeiro conceito tem um certo relacionamento inverso

com algum indivíduo pertencente ao segundo conceito, por exemplo, se todo filho tem ao menos um pai (e.g. $Child \sqsubseteq \exists fatherOf^{-}.Father$).

- ***queryAllValuesRestriction***. Este método recebe dois conceitos (e.g. *Father* e *Human*) e uma relação (e.g. *hasChild*) como entrada. Seu propósito é confirmar se todo indivíduo referente ao primeiro conceito pode apenas possuir um certo relacionamento com indivíduos pertecentes ao segundo conceito, por exemplo, se um pai só pode ter filhos que sejam humanos (e.g. $Father \sqsubseteq \forall hasChild.Human$).

É apresentado na Figura 27, a seguir, um fluxo para exemplificar o processo de execução do *CoreACQ*, desde a entrada de uma CQ através de sua *interface* pública até a resposta da validação. No exemplo, um sistema externo - isto é, de terceiros - realiza uma consulta para confirmar se a classe *Woman* (Mulher) é subclasse de *Human* (Humano), para isso utilizando o método *querySubclass* oferecido pela *interface* pública. A consulta ativa as funcionalidades do módulo *SUMO Inference Engine* (1), que inicia verificando se o mecanismo de *cache* foi habilitado, caso sim, tal mecanismo é executado (2) para tentar validar a CQ. Se a dedução não for efetuada, a consulta é pré-processada (3) com o propósito de identificar conceitos desconhecidos (4) e selecionar premissas (5) através dos conceitos da CQ, considerando, entretanto, que ambas as funções estejam previamente habilitadas. Em seguida, o *E Prover ATP* é executado para tentar validar a CQ por meio de axiomas da ontologia SUMO (6). Caso a dedução seja realizada, a prova obtida pode ser salva na ontologia *cache* (7) e fornecida como resposta para o sistema externo, caso contrário a resposta indica que nenhuma prova foi encontrada.

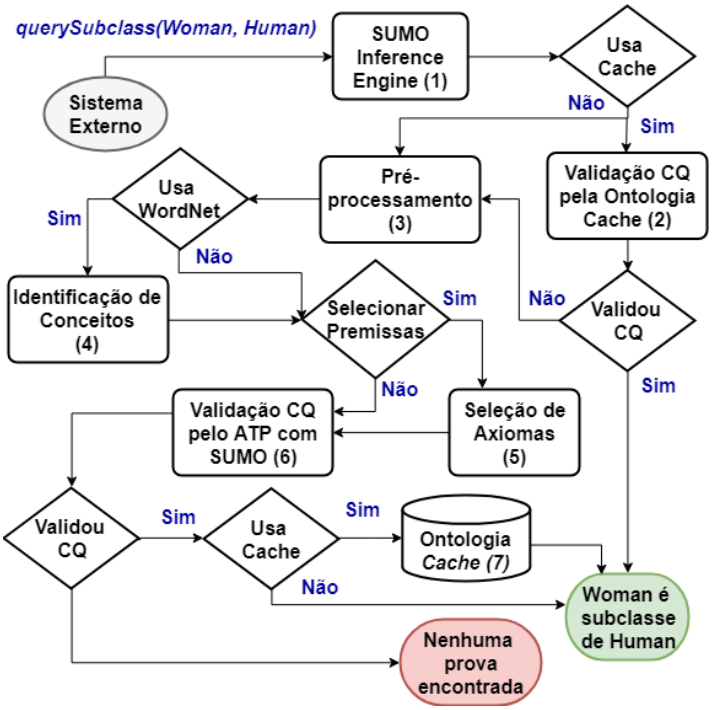

Figura 27 – Fluxo exemplificando a comunicação entre um sistema externo e o *CoreACQ*

3.2 Tecnologias

Nesta seção, as tecnologias adotadas para o desenvolvimento do *CoreACQ* foram apresentadas e discutidas. Particularmente, aqui é explicado como ocorre a manipulação da SUMO, ontologias OWL 2 DL, máquinas de inferência, provadores automáticos de teoremas e dicionários *WordNet*.

3.2.1 OWL API

A tecnologia OWL API (Application Program Interface[3], Interface de Programação de Aplicações) é uma biblioteca de código-aberto implementada na linguagem de programação Java. Seu objetivo é facilitar o projeto de sistemas relacionados à criação, manipulação e armazenamento de ontologias na linguagem OWL 2 DL. As suas principais funcionalidades são:

- Criação e atualização de ontologias em memória[4];
- Leitura e escrita de arquivos em vários formatos, por exemplo, RDF/XML e OWL/XML;
- Comunicação com motores de inferência (e.g. *Pellet* e *RACCOON*) para checagem de consistência e raciocínio sobre axiomas em OWL 2 DL;

Assim sendo, esta tecnologia é a responsável por toda operação de leitura e escrita em ontologias dentro do módulo ***Core OWL 2*** (Vide Seção 3.1.2). Consequentemente, fornece os recursos necessários para o manuseio da ontologia *Cache* dentro do módulo ***SUMO Inference Engine*** (Vide Seção 3.1.4).

Para exemplificar, quando a CQ "*Mulher é subclasse de Humano*" é validada por raciocínio sobre a SUMO, a tarefa de representar essa informação em OWL 2 DL é atribuída à biblioteca OWL API, que produz o código na sintaxe RDF/XML, conforme a Figura 28, a seguir.

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

<owl:Class rdf:about="#Mulher">
    <rdfs:subClassOf rdf:resource="#Humano"/>
</owl:Class>
```

Figura 28 – Código na sintaxe RDF/XML definindo que a classe Mulher é subclasse de Humano

[3] Em computação, API é um conjunto de rotinas e padrões estabelecidos para aplicativos executarem funcionalidades de um sistema sem precisar entender detalhes de sua implementação

[4] Em computação, memória pode ser entendida como um dispositivo que permite a um computador armazenar dados, de forma temporária ou permanente

3.2.2 Reasoners OWL 2 DL

No contexto da OWL 2, os motores de inferência desempenham um papel fundamental tratando-se de dedução de conhecimentos. Esse tipo de sistema objetiva utilizar axiomas conhecidos (declarados) para deduzir novos fatos a partir de outros e, inclusive, checar a consistência de uma ontologia de forma automática.

Aqui, nós utilizamos dois motores de inferência. O primeiro raciocinador, denominado *Pellet* (Vide Seção 2.2.1 do Capítulo 2), é uma biblioteca Java de código-aberto que implementa o método dos *Tableaux*[5] para inferências sobre OWL 2 DL (PARSIA; SIRIN, 2004). Na sua versão atual é qualificado para exercer vários serviços de raciocínio sobre ontologias como, por exemplo, consultas, checagem de consistência e classificação de hierarquia de conceitos.

O segundo raciocinador, denominado *RACCOON* (Vide Seção 2.2.2 do Capítulo 2), foi desenvolvido na linguagem de programação *C++*. Ele implementa o cálculo de conexões *ALC θ-CM* para verificar a consistência de ontologias OWL 2 DL com expressividade *ALC* (FILHO; FREITAS; OTTEN, 2017). Entretanto, esta funcionalidade pode ser usada para inferir um novo axioma seguindo duas tarefas: (I) Conhecimentos que contradizem o axioma são adicionados temporariamente na ontologia; (II) Em seguida verifica-se a consistência da ontologia, caso fique inconsistente então o axioma foi inferido, caso contrário nada é concluído.

Dessa forma, esse tipo de tecnologia é primordial para que o módulo ***SUMO Inference Engine*** execute raciocínio acerca dos conhecimentos da ontologia *Cache*, como visto na Seção 3.1.4.

3.2.3 JWI

A *MIT Java WordNet Interface* (JWI) é uma biblioteca Java desenvolvida por pesquisadores do *Massachusetts Institute of Technology*[6] (MIT, em tradução livre Instituto Tecnológico de Massachusetts). Seu propósito é simplificar o projeto de sistemas que acessam a *WordNet*, um dicionário semântico livre e público na lingua inglesa. Suas principais funções são em relação a consultas que procuram por significados, sinônimos, hipônimos, hiperônimos e, até, antônimos a respeito de uma palavra (conceito). Assim, esta tecnologia tem a capacidade de conectar-se às versões 1.6 até a 3.1 da *WordNet* (FINLAYSON, 2014). Portanto, a JWI é um importante recurso dentro do módulo ***Preprocessor*** (Vide Seção 3.1.3) devido a adquirir sinônimos e hiperônimos de conceitos desconhecidos pela SUMO.

[5] Na teoria da prova, o tableau semântico é um sistema de dedução e um procedimento de prova para fórmulas da lógica de primeira ordem

[6] http://web.mit.edu

3.2.4 E Prover ATP

O *E Prover* é um sistema ATP para FOL com operador de igualdade, seu projeto é código-aberto desenvolvido na linguagem de programação C, sua principal função é rastrear em uma base de conhecimento um conjunto de axiomas que prova uma sentença de entrada como verdadeira.

Sobre o formato de entrada e saída de dados, o *E Prover* faz uso da sintaxe desenvolvida com finalidade de apoiar o projeto TPTP (*Thousands of Problems for Theorem Provers*, Milhares de Problemas para Provadores de Teorema), que foi criado para prover o reuso de um conjunto de problemas pré-definido para testes em ATPs. Cada axioma é definido como uma fórmula particular contendo pelo menos identificador, categoria e a respectiva expressão lógica. Para exemplificar, a seguir é apresentada uma expressão (3.1) em FOL em conjunto com sua representação (3.2) na sintaxe TPTP:

$$\forall x Woman(x) \rightarrow Human(x) \tag{3.1}$$

$$fof(merge1, conjecture, (![X] : (Woman(X) => Human(X)))). \tag{3.2}$$

O primeiro argumento de uma fórmula em TPTP refere-se a um identificador único. Já o segundo argumento diz respeito a categoria, caso *axiom* (axioma) trata-se de uma afirmação sobre o domínio da ontologia, se *conjecture* (conjetura) então significa uma consulta (pergunta). O terceiro argumento é aquele que determina de fato a expressão lógica. Com relação à fórmula 3.2, o identificador é *merge1*, a categoria é *conjecture*, e a expressão lógica é "todo indivíduo que é mulher também tem como característica ser um humano".

São apresentadas na Tabela 1, a seguir, as expressões em FOL e suas respectivas representações na sintaxe TPTP, visando elucidar o formato de axiomas compatível com o *E Prover*.

Tabela 1 – Representação FO vs TPTP

Expressão em FO	Expressão em TPTP
A $\wedge$ B	A & B
A $\vee$ B	A | B
$\neg$ A	$\sim$ A
A $\rightarrow$ B	A => B
A $\leftrightarrow$ B	A <=> B
$\exists x$ A(x)	?[x]: A(x)
$\forall x$ A(x)	![x]: A(x)

Esta tecnologia (*E Prover*) constitui o módulo ***Core ATP*** (Vide Seção 3.1.1), atribuindo-se das operações de inferência sobre ontologias em FOL. Com isso, torna-se fundamental

para o módulo ***SUMO Inference Engine*** (Vide Seção 3.1.4) realizar raciocínio automático sobre a SUMO.

3.3 Uso do *CoreACQ*

Antes de inicializar o *CoreACQ*, a instalação do sistema *E Prover* no computador é obrigatória, para isso os arquivos e as instruções necessárias estão disponíveis em seu próprio site[7]. Ainda é necessário existir um arquivo no computador contendo os axiomas da ontologia SUMO na sintaxe TPTP. Caso seja cogitado o uso da *WordNet*, seus respectivos arquivos também devem estar acessíveis no computador. As instruções necessárias para a instalação do *CoreACQ* são discutidas em detalhes no Apêndice A.

A configuração do *CoreACQ* pode ser efetuada após a instalação do *E Prover*. O primeiro passo é fornecer a localização no computador dos arquivos referentes ao *E Prover*, base de conhecimento *SUMO-TPTP* e, se for o caso, *WordNet*. Neste momento, na hipótese de ser desejado, a função de seleção de premissas (Vide Seção 3.1.3) pode ser configurada por meio de um parâmetro que define o custo máximo de caminho. Do mesmo modo, a ontologia *Cache* (Vide Seção 3.1.4) pode ser configurada através da indicação do diretório[8] onde a mesma deve ser salva.

Depois da configuração do *CoreACQ*, ele está pronto para ser inicializado e exercer suas funcionalidades. Logo, todos os métodos do *framework* tornam-se acessíveis para executar a validação de diferentes tipos de CQs (Vide Seção 3.1.4).

3.4 Conclusão

Apresentamos neste capítulo as características do *framework* desenvolvido. A arquitetura do *CoreACQ* foi definida e a função de cada um de seus componentes foi explicada em detalhes. Por fim, o capítulo discutiu a respeito dos arquivos e informações necessários para a configuração do *CoreACQ*, possibilitando assim sua inicialização.

O próximo capítulo apresenta os experimentos executados para a avaliação do *CoreACQ*, além da investigação dos resultados alcançados.

[7] http://wwwlehre.dhbw-stuttgart.de/ sschulz/E/Download.html

[8] Em informática, diretório refere-se a uma estrutura usada para organizar arquivos em um computador, podendo ser um arquivo que contém referências, ou localizações, de outros arquivos

4 AVALIAÇÃO E RESULTADOS DO COREACQ

Esta dissertação apresentou até o momento *CoreACQ*, um *framework* para evoluir ontologias de domínio por validação de CQs através de raciocínio automático sobre a SUMO. Neste capítulo avaliamos e apresentamos resultados que demonstram a eficiência deste *framework* em realizar validações de diferentes tipos de CQs, assim conseguimos constatar sua capacidade em apoiar o desenvolvimento autônomo de uma ontologia de domínio.

Objetivando a avaliação do *CoreACQ*, executamos um experimento controlando a utilização do *framework*, manipulando as variáveis envolvidas conseguimos obter resultados satisfatórios que cumprem os objetivos definidos no Capítulo 1.

Apresentamos neste capítulo a elaboração e execução dos experimentos executados com *CoreACQ*, os resultados obtidos em cada teste e, também, aspectos a respeito de corretude e completude em termos de raciocínio automático. No final, é apresentado um cenário de utilização do *framework* como um componente do *Protégé*[1], que é uma ferramenta para o desenvolvimento de ontologias.

4.1 Avaliação do *CoreACQ*

A preparação do experimento foi baseada no modelo de cinco fases proposto por Wohlin et al (WOHLIN et al., 2012), sendo dividido em cinco atividades, conforme a seguir:

- **Definição**, onde devem ser especificados os aspectos mais importantes no experimento e a razão pela qual foi conduzido;
- **Planejamento**, que determina como o experimento será realizado;
- **Execução**, onde acontece a coleta dos dados para a avaliação na próxima atividade;
- **Análise e Interpretação**, onde os dados coletados na atividade **Execução** são análisados e interpretados para a geração dos resultados alcançados;
- **Apresentação**, última atividade que apresenta os resultados obtidos no experimento por meio de gráficos e tabelas;

As cinco atividades são apresentadas nas subseções a seguir, na mesma ordem em que foram apresentadas acima.

[1] https://protege.stanford.edu/

4.1.1 Definição

Para a definição do experimento foi adotado o paradigma *Goal Question Metric* (SOLINGEN et al., 2002) (GQM, em tradução livre Medidas para Questão Objetivo). GQM determina como formular o objetivo do estudo, as perguntas a serem respondidas e as medidas essenciais para produzir as respostas. À vista disso, a seguir são descritos esses três aspectos para a avaliação do *CoreACQ*.

Objetivo

Verificar a capacidade do *CoreACQ* em comunicar-se com o sistema *E Prover* para realizar inferências e validar CQs por meio de conhecimentos provenientes de versões da SUMO em FOL.

Perguntas

As perguntas a serem respondidas são:

$\mathbf{P}_1$. O *framework* realiza validações de CQs por meio de raciocínio automático sobre a ontologia SUMO em FOL?

$\mathbf{P}_2$. O *framework* realiza validações de CQs em um tempo hábil do ponto de vista das ferramentas da engenharia de ontologias?

Medidas

M_1. Cobertura (WIVES, 1999; MINTO; MURPHY, 2007): calcula-se através da divisão da quantidade de validações corretas pelo total de CQs que devem de fato ser inferidas pela base de conhecimento SUMO. O resultado permite avaliar se alguma CQ verdadeira não foi devidamente validada pelo *CoreACQ*.

M_2. Precisão (WIVES, 1999; MINTO; MURPHY, 2007): calcula-se através da divisão da quantidade de validações corretas pelo total de CQs processadas. Esta medida aponta se alguma CQ não é verdadeira - isto é, que não pode ser inferida pela SUMO - foi validada pelo *CoreACQ*.

M_3. Tempo de Resposta: é o tempo em segundos necessário para que uma CQ seja validada pelo *framework*. Seu cálculo é obtido através do maior tempo necessário para processar uma validação. Essa medida é importante, uma vez que as ferramentas da engenharia de ontologias não podem esperar uma resposta por tempos exorbitantes. Taylor, Dennis e Cummings (2013) argumentam que os usuários se acostumaram a tempos de resposta toleráveis até um certo ponto - sua pesquisa sugere entre 7 e 11 segundos - após o qual um sistema perde a atenção dos seus usuários.

4.1.2 Planejamento

A versão escolhida da SUMO para o experimento precisa estar de acordo com a linguagem de representação de conhecimento compatível com o sistema *E Prover*. Portanto a versão deve estar previamente representada em FOL, evitando assim responsabilidades com conversões entre KIF e FOL.

Contexto. O objetivo do experimento é analisar a viabilidade do uso do *CoreACQ* em ferramentas da engenharia de ontologias para validar CQs automaticamente por conhecimentos derivados de versões da SUMO em FOL. Para isso, foi escolhida a *Adimen-SUMO v2.6*[2] para prover os conhecimentos básicos para o procedimento de validação utilizado pelo *framework*, mais detalhes na Seção 4.1.2.1.

Participantes. Os participantes desse estudo são CQs relacionadas à *Adimen-SUMO*, a ontologia de topo escolhida para o experimento.

Hipóteses nulas. São hipóteses a serem rejeitadas pelo pesquisador. No experimento, as hipóteses nulas determinam que o *CoreACQ* não está capacitado para realizar validações de CQs de forma autônoma por racioínio sobre a SUMO, ou requer de um tempo excessivo para exercer tal atividade. Sendo assim, as seguintes hipóteses foram elaboradas:

$$\mathbf{H}_{n1} \text{ : cobertura do } \mathit{CoreACQ} < 1.0$$

$$\mathbf{H}_{n2} \text{ : precisão do } \mathit{CoreACQ} < 1.0$$

$$\mathbf{H}_{n3} \text{ : tempo de resposta do } \mathit{CoreACQ} > 10 \text{ segundos}$$

Hipóteses alternativas. São hipóteses com o objetivo de rejeitar as hipóteses nulas. As hipóteses alternativas são:

$$\mathbf{H}_{a1} \text{ : cobertura do } \mathit{CoreACQ} = 1.0$$

$$\mathbf{H}_{a2} \text{ : precisão do } \mathit{CoreACQ} = 1.0$$

$$\mathbf{H}_{a3} \text{ : tempo de resposta do } \mathit{CoreACQ} <= 10 \text{ segundos}$$

Variáveis independentes. São variáveis controladas e manipuladas durante o experimento. Então, as variáveis independentes referem-se às validações de CQs feitas pelo *CoreACQ*, o uso do mecanismo de *cache*, seleção de premissas e a utilização da *WordNet*.

Variáveis dependentes. São os objetos de estudo do experimento, seus valores variam com as mudanças feitas nas veriáveis independentes. Logo, as variáveis dependentes são cobertura, precisão e tempo de resposta das validações feitas pelo *CoreACQ*.

Validade interna. Verifica se os resultados alcançados são causados pelo tratamento realizado no experimento, ou seja, se a modificação das variáveis independentes de fato refletiu nas variáveis dependentes. No caso de raciocínio sobre axiomas em FOL, o tempo de resposta de sistemas como *E Prover* é ampliado conforme o tamanho da base de conhecimento, portanto, ativar a seleção de premissas para reduzir a quantidade de axiomas a serem usados na etapa de raciocínio deve resultar uma diminuição no tempo das validações. Além disso, considerar o uso do procedimento de *cache* também deve agilizar a resposta do *framework*, dado que evita repetição de atividades no *E Prover*. A utilização

[2] http://adimen.si.ehu.es/web/AdimenSUMO

da *WordNet* para enriquecer o vocabulário da *Adimen-SUMO* também deve influenciar os resultados. Por intermédio de um dicionário léxico vem a ser possível a validação de CQs que envolvem conceitos não conhecidos pela SUMO.

Validade externa. Verifica se o experimento pode ser generalizado, isto é, a facilidade do mesmo estudo ser repetido com outras versões da SUMO. Os resultados são influenciados pelos conhecimentos da ontologia de topo usada pelo *CoreACQ*. No entanto, como a ontologia *Adimen-SUMO v2.6* é livre e de código aberto, a replicação e a validade do experimento são consideradas suficientes para qualificar o estudo realizado.

Validade de conclusão. Determina se o experimento é capaz de produzir conclusões corretas sobre as relações entre o tratamento das variáveis e os resultados alcançados. As conclusões serão criadas por meio da análise comparativa das medidas de cobertura, precisão e tempo de resposta das validações realizadas pelo *CoreACQ*. As validações serão executadas através de um conjunto de CQs extraído da versão SUMO utilizada no experimento.

4.1.2.1 *Versão SUMO utilizada no experimento*

A versão da SUMO escolhida para o experimento foi *Adimen-SUMO v2.6*, uma ontologia de topo que pode ser usada apropriadamente por provadores de teoremas (e.g. *E Prover*) para raciocínio automático. Por meio dela, o experimento pode lidar com problemas reais de inferência sobre axiomas em FOL para evoluir ontologias de domínio de forma autônoma.

Com a Adimen-SUMO, torna-se viável simular a validação de CQs fazendo uso do *CoreACQ* na hipótese de ontologias de domínio que necessitam de conhecimentos disponíveis em ontologias de topo.

4.1.2.2 *Instrumentação*

A simulação foi produzida tendo como entrada um conjunto de CQs obtido por meio de um processo semi-automático de extração de conhecimentos sobre a ontologia *Adimen-SUMO v2.6*, disponível em http://adimen.si.ehu.es/web/AdimenSUMO.

Na primeira fase da extração, os axiomas da ontologia *Adimen-SUMO* foram examinados e selecionados de acordo com seus significados. Entretando somente foram considerados aqueles relacionados explicitamente com subsunção, disjunção, equivalência, relacionamentos e relacionamentos inversos. A seguir estão alguns exemplos do resultado dessa etapa:

$$subclass(Woman, Human) \tag{4.1}$$

$$disjoint(DomesticAnimal, Human) \tag{4.2}$$

$$\forall x\,[instance(x, Eye) \rightarrow \exists y\,(instance(y, Head) \wedge part(x, y))] \tag{4.3}$$

A CQ 4.1 define que todos os indivíduos da classe mulher também estão contidos na classe humano, ou seja, toda mulher é um ser humano. A CQ 4.2 refere-se ao fato de que um humano não pode ser considerado um animal doméstico. Já a CQ 4.3 expressa que todo indivíduo da classe olho faz parte de algum indivíduo da classe cabeça, ou seja, todo olho faz parte de uma cabeça. Dessa maneira, a primeira parte da amostra, contendo 958 CQs, foi construída por axiomas explicitamente declarados na *Adimen-SUMO*.

O experimento precisa tratar CQs que não estão explicitamente declaradas, ou seja, cuja dedução necessita de mais de um axioma da ontologia. Por esse motivo uma nova tarefa da extração foi executada através de regras de inferência sobre as CQs construídas na fase anterior. Para exemplificar, sabe-se que uma mulher é um humano (4.1) e todo humano não é um animal doméstico (4.2), logo deduz-se que toda mulher não é um animal doméstico, conforme a CQ 4.4, a seguir:

$$disjoint(DomesticAnimal, Woman) \tag{4.4}$$

Sendo assim, a utilização da regra de inferência de subsunção possibilitou a criação de 1681 CQs que não estão representadas explicitamente na *Adimen-SUMO*. Essa nova porção expande a complexidade da amostra em relação a tarefa de raciocínio feita pelo *E Prover*.

A SUMO, como qualquer outra ontologia, está restrita ao seu vocabulário - isto é, os conceitos declarados em sua base de conhecimento. Por isso ainda foi realizada uma terceira fase de extração com o suporte da *WordNet*. A colaboração desse dicionário léxico viabiliza a aquisição do significado de um conceito desconhecido pela ontologia de topo, visando descobrir uma relação com algum conceito conhecido. Por exemplo, de acordo com *WordNet* o termo *Human Being* (Ser Humano) é um sinônimo para *Human* (Humano). Assim, as CQs 4.1 e 4.2 podem ser reescritas considerando-se *Human Being* em vez de *Human*, conforme as CQs 4.5 e 4.6, a seguir. Desta forma, nesta última etapa de extração foram construídas mais 240 CQs para a amostra do experimento.

$$subclass(Woman, HumanBeing) \tag{4.5}$$

$$disjoint(DomesticAnimal, HumanBeing) \tag{4.6}$$

As três etapas de extração foram realizadas durante um período de três meses, sendo os resultados de cada etapa reunidos em uma única amostra para o experimento. Ao final

foi construído um conjunto contendo 2879 CQs, distribuídas de acordo com os seus tipos, conforme a Tabela 2, a seguir:

Tabela 2 – Distribuição da amostra de CQs com relação aos seus tipos

Tipo de CQ	***Quantidade***
$A \sqsubseteq B$	991
$A \sqsubseteq \neg B$	550
$A \equiv B$	32
$A \sqsubseteq \exists prop.B$	713
$A \sqsubseteq \exists prop^{-}.B$	518
$A \sqsubseteq \forall prop.B$	75

4.1.3 Execução

O experimento foi dividido em cinco segmentos com o objetivo de avaliar várias configurações do *CoreACQ*, sendo cada segmento tratado como um experimento específico. O primeiro faz uso da configuração principal do *framework*, isto é, considera somente a função de inferência sobre a *Adimen-SUMO* para o processo de validação por meio do *E Prover*. Assim, as 2879 CQs da amostra de teste foram passadas individualmente como entrada para o *framework* e, após sua execução, os resultados e tempos de resposta foram armazenados em um log de dados[3]. Apresentamos na Tabela 3, a seguir, a quantidade de CQs validadas de acordo com o tempo de resposta do primeiro experimento.

Tabela 3 – Resultado do primeiro experimento com a quantidade de CQs validadas por tempo de resposta

#CQs / Tempo de Resposta	**Quantidade de Deduções**
$0s <$ Tempo $< 0.1s$	0
$0.1s <$ Tempo $< 1s$	2097
$1s <$ Tempo $< 10s$	331
$10s <$ Tempo $< 100s$	178
$100s <$ Tempo $< 600s$	33
CQs não validadas	240

Na Tabela 3 observam-se várias CQs não validadas, a causa disso refere-se a ocorrência de conceitos não conhecidos - isto é, não definidos no vocabulário da SUMO. Com o objetivo de resolver esse problema, o segundo experimento foi realizado com o apoio de um dicionário léxico. Agora o *CoreACQ* é configurado para aplicar raciocínio automático

[3] Em computação, log de dados é um arquivo utilizado para descrever eventos relevantes sobre um sistema computacional

usando o *E Prover* e, ainda, empregar a *WordNet* para descobrir o significado de conceitos desconhecidos. O resultado dessa mudança pode ser verificado na Tabela 4, a seguir:

Tabela 4 – Resultado do segundo experimento com a quantidade de CQs validadas por tempo de resposta

#CQs / Tempo de Resposta	**Quantidade de Deduções**
$0s <$ Tempo $< 0.1s$	0
$0.1s <$ Tempo $< 1s$	2313
$1s <$ Tempo $< 10s$	355
$10s <$ Tempo $< 100s$	178
$100s <$ Tempo $< 600s$	33
CQs não validadas	0

Na Tabela 4 é importante notar a quantidade (211) de validações com tempo de resposta superior a 10 segundos. Sendo assim, o terceiro experimento foi realizado tendo em conta as funções de comunicação com o *E Prover*, consultas ao *WordNet* e, também, a seleção de premissas implementada pelo *CoreACQ* (Vide Seção 3.1.3 do Capítulo 3). Sobre o parâmetro de custo máximo de caminho foram examinados, respectivamente, custos cinco (C.M. 5), quatro (C.M. 4), três (C.M. 3), dois (C.M. 2) e um (C.M. 1). Devido à quantidade de experimentos, custos de caminho maiores do que cinco não foram analisados, o que podemos deixar como trabalhos futuros. Os resultados obtidos nesse experimento são apresentados na Tabela 5, a seguir.

Tabela 5 – Resultado do terceiro experimento com a quantidade de CQs validadas por tempo de resposta

	Quantidade de Deduções				
#CQs / Tempo Resposta	***C.M. 5***	***C.M. 4***	***C.M. 3***	***C.M. 2***	***C.M. 1***
$0s <$ Tempo $< 0.1s$	50	60	130	258	1984
$0.1s <$ Tempo $< 1s$	2803	2793	2723	2604	895
$1s <$ Tempo $< 10s$	26	26	26	17	0
$10s <$ Tempo $< 100s$	0	0	0	0	0
$100s <$ Tempo $< 600s$	0	0	0	0	0
CQs não validadas	0	0	0	0	0

Um sistema computacional está sujeito a repetição de uma mesma atividade em momentos distintos. Portanto existe a possibilidade do *CoreACQ* validar uma CQ exercendo as mesmas tarefas de outras validações, como discutido na Seção 3.1.4 do Capítulo 3. Então, o quarto experimento foi executado tendo em consideração as funções de comunicação com o *E Prover*, consultas ao *WordNet* e, também, do procedimento de *cache* com *RACCOON*. A finalidade agora é otimizar o tempo de validação de uma CQ por infor-

mações sobre validações já processadas anteriormente. Na Tabela 6, a seguir, é mostrado o resultado desse experimento.

Tabela 6 – Resultado do quarto experimento com a quantidade de CQs validadas por tempo de resposta

#CQs / Tempo de Resposta	**Quantidade de Deduções**
$0s <$ Tempo $< 0.1s$	887
$0.1s <$ Tempo $< 1s$	1726
$1s <$ Tempo $< 10s$	203
$10s <$ Tempo $< 100s$	46
$100s <$ Tempo $< 600s$	17
CQs não validadas	0

Para finalizar é preciso investigar o comportamento do *CoreACQ* levando em consideração a entrada de CQs não verdadeiras. No quinto experimento, um grupo de 500 CQs falsas (e.g. $Woman \sqsubseteq Man$) foi produzido por intermédio da amostra coletada na instrumentação (Vide Seção 4.1.2.2). Em seguida, cada uma foi passada como entrada para o *framework* realizar sua validação. O resultado desse experimento demonstrou que independente da configuração adotada - isto é, uso da *WordNet*, *cache* ou seleção de premissas - nenhuma das CQs falsas foi validada pelo *framework*.

Todos os experimentos foram executados em um *notebook* com 8GB de memória RAM (DDR3 1600MHz), 1TB de espaço em disco rígico (5400rpm), processador Intel Core i7 (4 núcleos) e sistema operacional Ubuntu 16.04.3 LTS.

4.1.4 Análise e interpretação

Logo após a execução de cada experimento, os resultados foram agrupados para o cálculo das medidas de cobertura (M_1), precisão (M_2) e tempo de resposta (M_3) das validações de CQs realizadas pelo *framework* desenvolvido. A análise foi realizada com o recurso de tabelas e gráficos que compreendem os valores das medidas obtidos pelos experimentos.

Visando o melhor entendimento dos resultados e gráficos, a partir deste momento serão adotados nomes para distinguir as diferentes configurações do *CoreACQ* em cada um dos experimentos. É apresentada na Tabela 7, a seguir, a nomenclatura a ser considerada, o símbolo (X) indica que a funcionalidade é utilizada, já o símbolo (-) aponta o contrário.

Avaliando a capacidade do *CoreACQ* em executar validações: esta análise foi realizada no intuito de verificar se o *framework* desempenha corretamente o seu papel fundamental, ou seja, se é capaz de validar todas as CQs verdadeiras investigadas no experimento.

Na Figura 29 apresentamos a cobertura do *CoreACQ* em relação as suas configurações na realização dos experimentos. Observa-se que a configuração E não alcançou o valor

Tabela 7 – Nomenclatura das configurações do *CoreACQ* empregues nos experimentos

Nomenclatura	***E Prover***	***WordNet***	***Seleção de Premissas***	***Cache***
E	X	-	-	-
E+W	X	X	-	-
E+W+CA	X	X	-	X
E+W+C5	X	X	X (Custo Máximo 5)	-
E+W+C4	X	X	X (Custo Máximo 4)	-
E+W+C3	X	X	X (Custo Máximo 3)	-
E+W+C2	X	X	X (Custo Máximo 2)	-
E+W+C1	X	X	X (Custo Máximo 1)	-

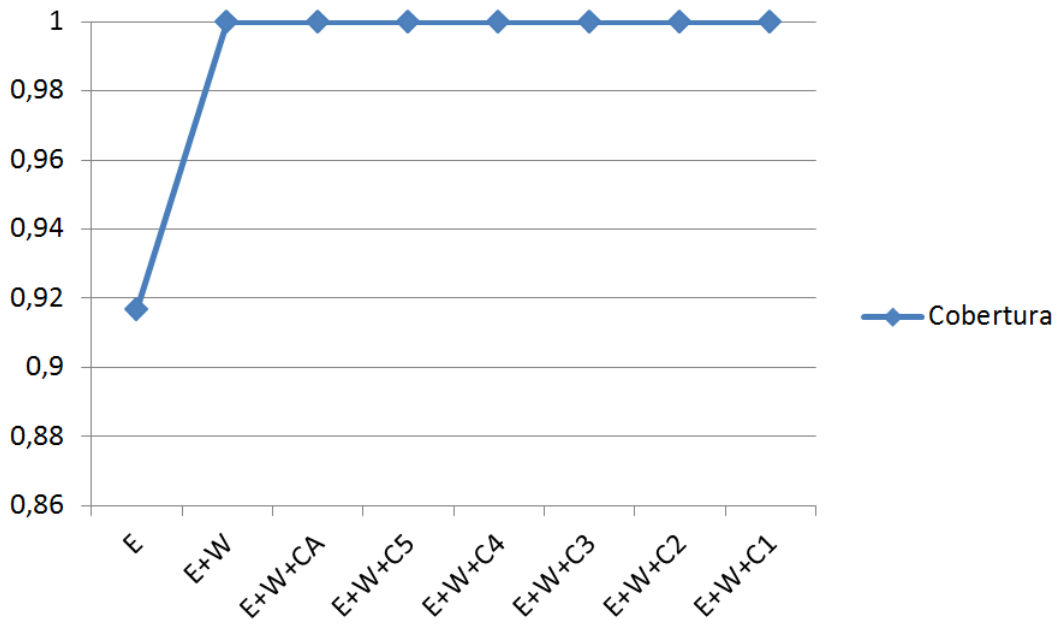

Figura 29 – Gráfico comparativo da cobertura das validações de CQs realizadas por cada configuração do *CoreACQ*

máximo desejado, mas isso ocorreu devido a conceitos desconhecidos pela *Adimen-SUMO*, e não por falhas ou erros de implementação. A partir da utilização da *WordNet* esse problema foi solucionado, com a cobertura chegando ao valor esperado (1.0) para comprovar a hipótese $\mathbf{H}_{a1}$ (Vide Seção 4.1.2).

Avaliando a habilidade do *CoreACQ* em não validar CQs falsas: esta análise foi feita visando garantir que o processo de raciocínio exercido pelo *framework* é confiável, isto é, baseia-se sobretudo nos axiomas da *Adimen-SUMO*.

Na Figura 30, a seguir, é indicada a precisão do *CoreACQ* levando em conta as suas configurações. À vista disso, cada configuração atingiu a precisão esperada (1.0) para comprovar a hipótese $\mathbf{H}_{a2}$, isto é, nenhuma validação falsa foi deduzida pelo *framework*.

Avaliando a viabilidade do *CoreACQ* em ser usado por ferramentas da

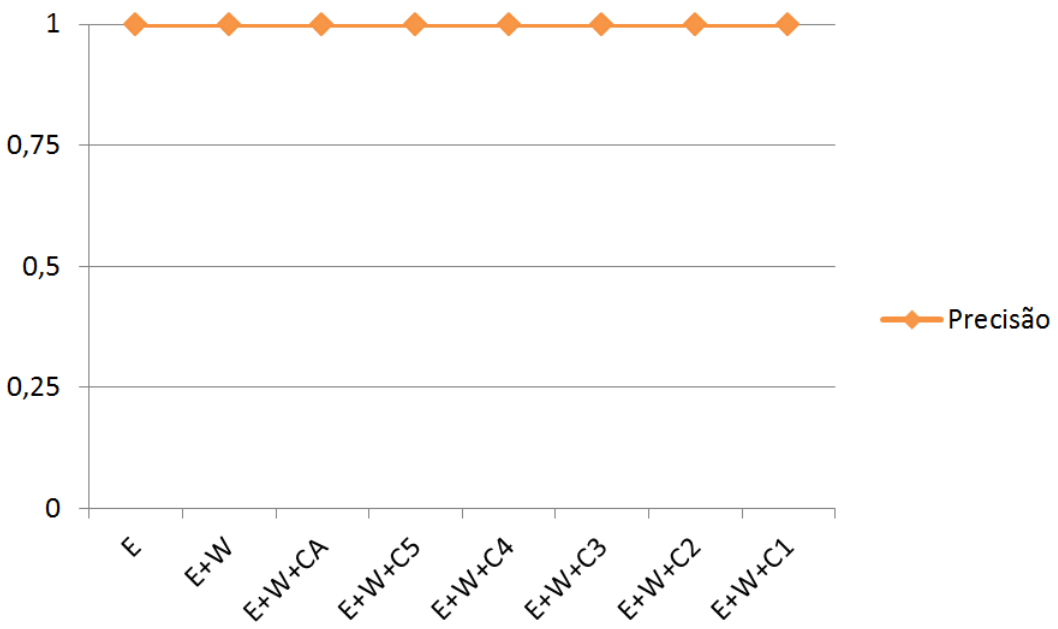

Figura 30 – Gráfico comparativo da precisão das validações de CQs realizadas por cada configuração do *CoreACQ*

engenharia de ontologias: a análise agora objetiva averiguar o tempo de resposta do *framework* necessário para deduzir uma validação e fornecer o resultado.

São apresentados nas Figuras 31 e 32, respectivamente, os tempos médio e máximo observados no processo de validação de CQs tendo em vista as configurações do *CoreACQ*. Todavia, os gráficos não expõem de modo claro e evidente quais foram os melhores resultados. Em razão disso, também é apresentado na Tabela 8 os tempos de resposta alcançados por cada configuração, expressando principalmente a influência da seleção de premissas - ou seja, a redução da base de conhecimento - no desempenho do *framework*.

Tabela 8 – Tempo médio e o tempo máximo das respostas nas validações de CQs por configuração do *CoreACQ*

	Validação de CQs Verdadeiras	
Configuração do *CoreACQ*	**Tempo Médio das Respostas**	**Tempo Máximo de Resposta**
E	5.386s	571.978s
E+W	5.042s	571.980s
E+W+CA	2.693s	476.022s
E+W+C5	0.224s	5.024s
E+W+C4	0.218s	5.052s
E+W+C3	0.213s	3.872s
E+W+C2	0.186s	1.598s
E+W+C1	0.101s	0.454s

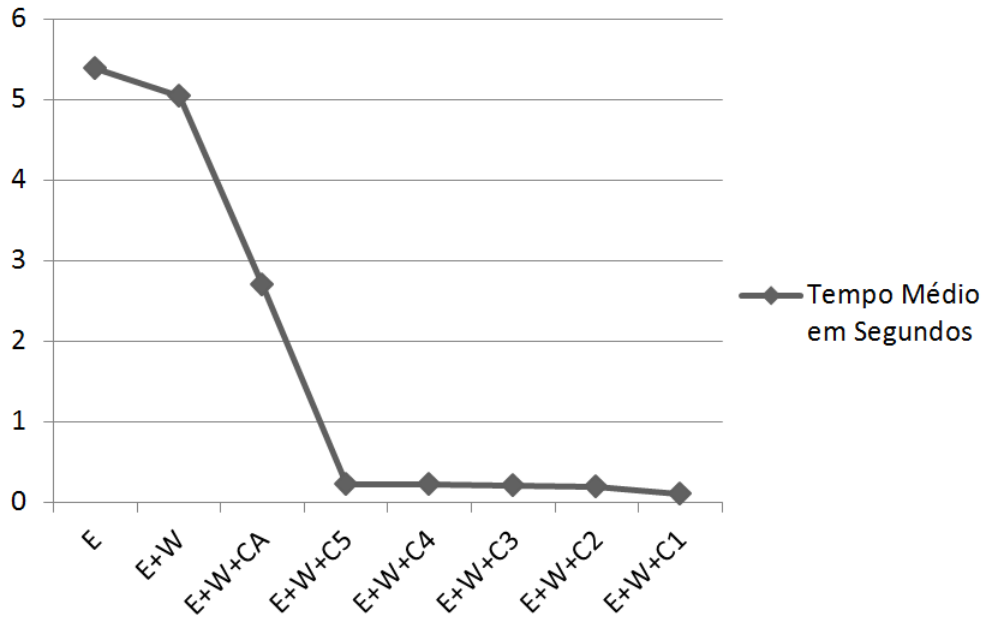

Figura 31 – Gráfico comparativo do tempo médio de resposta das validações de CQs realizadas por cada configuração do *CoreACQ*

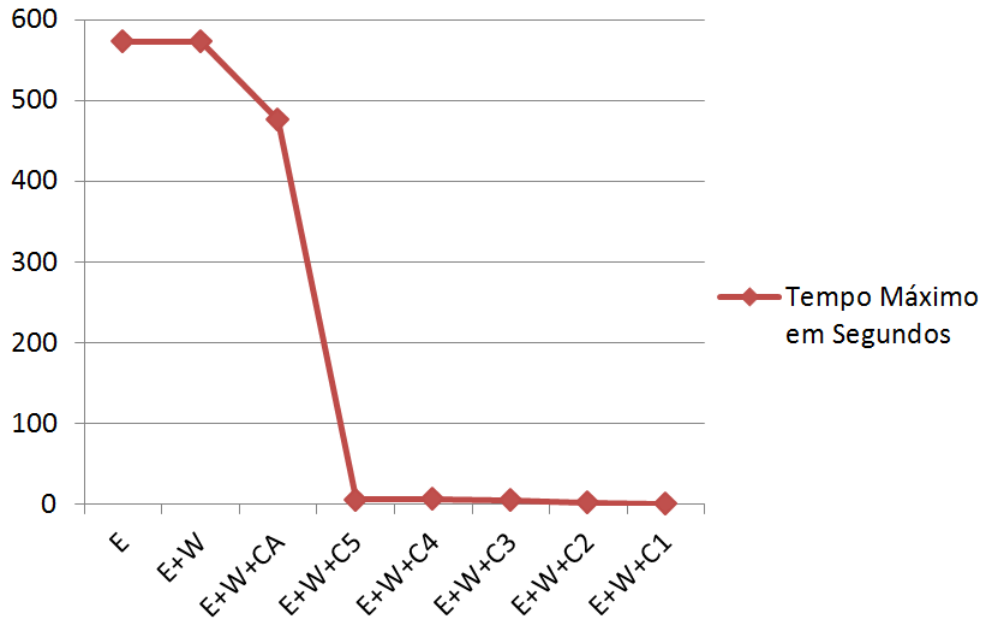

Figura 32 – Gráfico comparativo do tempo de resposta máximo das diferentes configurações do *CoreACQ*

Em relação ao tempo médio, é importante constatar que todas as configurações atingiram resultados para confirmar a hipótese $\mathbf{H}_{a3}$, como discutido na Seção 4.1.2. Tratando-se do tempo máximo, nota-se a incidência de longos períodos de tempo para *CoreACQ* realizar uma única validação, por exemplo 571 segundos - ou seja, aproximadamente 10 minutos. No entanto, esse tempo foi diminuído para praticamente 5 segundos ou menos por meio da seleção de premissas implementada pelo *framework*, confirmando assim $\mathbf{H}_{a3}$.

Avaliando simultaneamente as medidas por configuração do *CoreACQ*: é necessário verificar se o *framework* é capaz de finalizar cada validação em até 10 segundos, isto é, se satisfaz as hipóteses H_{a1} e H_{a3} ao mesmo tempo.

Na Figura 33, a seguir, é apresentada a cobertura acumulada por limite de tempo de validação (dedução) considerando-se cada configuração do *framework*. Seu gráfico possibilita identificar quais configurações obtiveram a cobertura máxima - ou seja, validação de todas as CQs verdadeiras - respeitando o limite máximo de 10 segundos para finalizar cada validação.

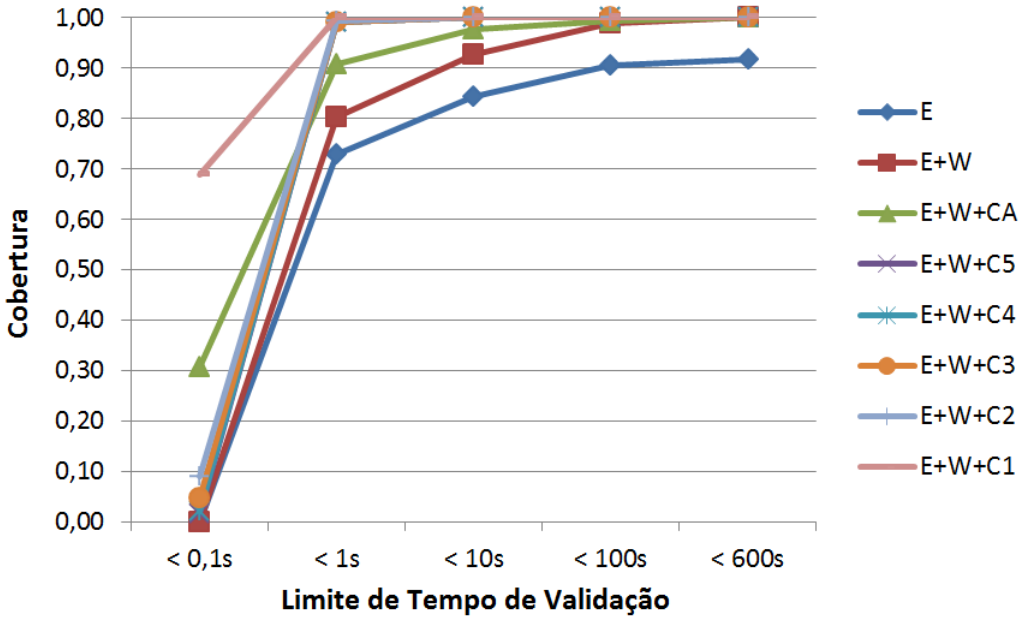

Figura 33 – Gráfico comparativo da cobertura acumulada por tempo de validação em cada configuração do *CoreACQ*

Apresentamos na Tabela 9, a seguir, as mesmas informações da Figura 33 em um formato tabular[4]. Como pode ser visto, a função de seleção de premissas tornou o *CoreACQ* capaz de validar cada CQ verdadeira em menos de 10 segundos. A existência de CQs cuja tarefa de dedução requer um longo período de tempo é justificável, uma vez que inferência sobre FOL pressupõe a utilização de um sistema ATP (e.g. *E Prover*). No geral, considerando o auxílio da *WordNet*, o *framework* alcançou uma cobertura de pelo menos 0.93 cumprindo o tempo estabelecido pela hipótese $\mathbf{H}_{a3}$.

[4] Formato tabular pode ser entendido como um padrão onde informações são apresentadas na forma de uma tabela, com linhas e colunas

Tabela 9 – Tabela comparativo da cobertura acumulada por tempo de resposta das validações realizadas por cada configuração do *CoreACQ*

	Cobertura Acumulada das Validações de CQs Realizadas pelo *CoreACQ* por Configuração							
Limite do Tempo de Resposta	***E***	***E+W***	***E+W+CA***	***E+W+C5***	***E+W+C4***	***E+W+C3***	***E+W+C2***	***E+W+C1***
100 milisegundos	0.00	0.00	0.31	0.02	0.02	0.05	0.09	0.69
1 segundo	0.73	0.80	0.91	0.99	0.99	0.99	0.99	1.00
10 segundos	0.84	0.93	0.98	1.00	1.00	1.00	1.00	1.00
100 segundos	0.91	0.99	0.99	1.00	1.00	1.00	1.00	1.00
10 minutos	0.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00

4.2 Corretude e Completude

Nos experimentos realizados e apresentados nas seções anteriores deste capítulo foi mostrado o desempenho do *CoreACQ* no que diz respeito a validar CQs, porém questões importantes sobre as propriedades de corretude e completude necessitam ser esclarecidas. Um sistema dedutivo é dito correto se tudo que ele prova de fato é uma consequência lógica da ontologia. O teorema da corretude nos garante que tudo que provamos pelo sistema é correto no que se refere à semântica. Isto é, assegura que um sistema só prova conjecturas (e.g. CQs) semanticamente corretas em relação à base de conhecimento. Por outro lado, um sistema é dito completo quando é capaz de provar tudo que é semanticamente válido, ou seja, o teorema da completude nos garante que o sistema encontra uma prova para qualquer conjectura implicada logicamente pela ontologia. Assim, é preciso investigar o *CoreACQ* em detalhes quanto aos teoremas da corretude e completude, o que podemos deixar como trabalhos futuros.

O *CoreACQ* em sua configuração mais simples delega todas as tarefas de raciocínio para o *E Prover*. Esse provador de teoremas implementa *Superposition Calculis* (em tradução livre Cálculo por Superposição), que foi demonstrado como correto e completo para FOL. Outra configuração do *CoreACQ* emprega um procedimento de *cache* por raciocínio sobre ontologia OWL 2 DL através do motor de inferência *RACCOON* que implementa ALC θ-CM, também demonstrado como correto e completo. Portanto, o *CoreACQ* herdou essas duas propriedades devido ao processo de raciocínio ser delegado integralmente para sistemas que satisfazem os dois teoremas.

A terceira configuração do *CoreACQ* faz uso de uma função para seleção de premissas que busca delimitar axiomas e reduzir a base de conhecimento com o objetivo de diminuir o tempo de resposta (validação) de uma conjectura (CQ) - conforme Seção 3.1.3 do Capítulo 3. Entretanto é necessário analisar o impacto dessa função a respeito de corretude e completude. Para isso consideremos uma ontologia O e uma conjectura falsa α, isto é, O não implica logicamente α ($O \not\models \alpha$). Por hipótese, vamos supor ainda que a etapa de delimitação resultou O', onde O' é um subconjunto de axiomas de O ($O' \sqsubseteq O$). Também consideremos que *CoreACQ* não é correto, ou seja, existe um grupo de premissas ($\{P_1, P_2, ..., P_n\}$) contidas em O' ($\{P_1, P_2, ..., P_n\} \sqsubseteq O'$) tal que implicam logicamente α ($\{P_1, P_2, ..., P_n\} \models \alpha$). Sendo assim, temos:

$$O \not\models \alpha \tag{4.7}$$

$$O' \subseteq O \tag{4.8}$$

$$\{P_1, P_2, ..., P_n\} \subseteq O' \tag{4.9}$$

$$\{P_1, P_2, ..., P_n\} \models \alpha \tag{4.10}$$

$$\tag{4.11}$$

Observando as afirmações (4.9) e (4.10) pode-se concluir que $O' \models \alpha$, em razão de O' conter as premissas $\{P_1, P_2, ..., P_n\}$. Já as afirmações (4.8), (4.9) e (4.10) implicam, por transitividade, que $\{P_1, P_2, ..., P_n\} \subseteq O$, com isso também é possível concluir que $O \models \alpha$. Entretando essa última conclusão contradiz a afirmação (4.7), ou seja, admitir a hipótese de que *CoreACQ* não satisfaz o teorema da corretude resultou em uma contradição.

O uso de um processo estritamente procedimental para delimitar os axiomas de uma ontologia na verdade afeta a propriedade de completude. Logo, as questões prepoderantes a serem elucidadas são: (1) *CoreACQ* proporciona perdas em relação à dedução de conjecturas semanticamente válidas?; e (2) Qual uma estimativa de ganhos e perdas provocados pelo *CoreACQ* no que diz respeito a raciocínio automático?. A estratégia para responder essas indagações é fazer uso de uma metodologia semelhante à utilizada na avaliação da ontologia *Adimen-Sumo*.

Foi desenvolvido por Álvez, Lucio e Rigau (2015) um conjunto de questões de competência com o intuito de avaliar a *Adimen-SUMO* em termos de completude. Por intermédio de um sistema ATP, eles realizaram um experimento para verificar quais CQs são inferidas por meio de raciocínio sobre sua ontologia. O resultado mostrou que várias CQs semanticamente válidas ficaram sem resposta em um *timeout*[5] de 600 segundos e, ainda, o tempo médio de dedução ficou abaixo de 60 segundos. Diante disso vem a ser viável a reprodução desse experimento no contexto do *CoreACQ*, no entanto considerando-se o *timeout* de 60 segundos, uma vez que corresponde ao tempo médio citado. Na Tabela 10, a seguir, os novos resultados obtidos são apresentados, indicando os custos máximos de caminho considerados, o número de CQs verdadeiras (P) que foram deduzidas, o tempo médio de execução ($t(P)$), e o número de CQs (U) cuja execução atingiu o *timeout* estabelecido.

Tabela 10 – Avaliação do *CoreACQ* em relação a completude e corretude de sua função de delimitação de axiomas

	CQs Válidas (1.000)		
Sistema	***P***	***U***	***t(P)***
E Prover	508	492	12.46 *s.*
CoreACQ com Custo Máximo 5	638	362	1.24 *s.*
CoreACQ com Custo Máximo 4	639	361	1.23 *s.*
CoreACQ com Custo Máximo 3	631	369	0.89 *s.*
CoreACQ com Custo Máximo 2	621	379	0.51 *s.*
CoreACQ com Custo Máximo 1	461	539	0.20 *s.*

Fazendo uso somente do *E Prover* foi possível realizar 508 deduções, faltando um total de 492 respostas. Em relação ao *CoreACQ* com delimitação de axiomas e custo máximo 1, foram obtidas 461 deduções, resultado inferior se comparado ao primeiro. Então

[5] Em informática, *timeout* pode ser definido como o tempo máximo permitido para que uma tarefa seja finalizada.

percebe-se uma perda de aproximadamente 9% de inferências válidas. Por outro lado, a delimitação com custo máximo 2 conseguiu 621 deduções, melhorando o processo de raciocínio automático tendo em vista quantidade de respostas em até 60 segundos. Os resultados continuaram melhorando com custo máximo 3 e custo máximo 4, respectivamente, 631 e 639 inferências. Já com custo máximo 5 houve um pequeno retrocesso no número de deduções, alcançando 638 respostas. Isso ocorreu devido à etapa de delimitação transmitir para o *ATP* grupos de axiomas cada vez maiores, causando tempos de execução superiores à 60 segundos. Sobre o tempo médio, em segundos, de execução por CQ, observa-se uma queda de 12.46 segundos (*E Prover*) para, respectivamente, 1.24 (*Custo Máximo* 5), 1.23 (*Custo Máximo* 4), 0.89 (*Custo Máximo* 3), 0.51 (*Custo Máximo* 2) e 0.20 (*Custo Máximo* 1) segundos. Assim este experimento demonstra que *CoreACQ* pode alcançar os melhores resultados em cenários com pequenos *timeouts*.

Na próxima seção é apresentado um cenário de utilização do *CoreACQ* dentro de uma ferramenta de desenvolvimento de ontologias conhecida como *Protégé*.

4.3 Cenário de Utilização

Nesta seção é apresentada uma aplicabilidade do *CoreACQ* na ferramenta conhecida como *Protégé*, que é um renomado sistema na área da engenharia de ontologias.

A Subseção 4.3.1, a seguir, aborda o *Protégé* e suas características. Na Subseção 4.3.2 é discutida a criação de um *plugin*[6] necessário para possibilitar as funções do *CoreACQ* dentro da arquitetura do *Protégé*. Por fim, a Subseção 4.3.3 demonstra o funcionamento e os resultados do *plugin* produzido.

4.3.1 Protégé

O projeto *Protégé* teve início nos anos 80 pela Universidade de Stanford[7]. Ao passar dos anos recebeu várias atualizações e melhorias, tornando-se atualmente uma ferramenta para a construção de ontologias e sistemas baseados em conhecimento amplamente utilizada. De acordo com Musen (2015), *Protégé* tornou-se a ferramenta de desenvolvimento de ontologias mais utilizada para a criação de ontologias, sendo sua equipe de desenvolvedores reconhecida com o Prêmio "Dez Anos"[8] na Conferência Internacional da Web Semântica, em outubro de 2014.

Em sua versão 5, o *Protégé* é um editor de ontologias e um sistema de gerenciamento de conhecimento, desenvolvido em Java, livre e de código-aberto. Seu maior propósito é fornecer uma interface gráfica para o usuário construir sua ontologia sem a necessidade de manipular diretamente o código OWL. Além disso, também permite o uso de motores

[6] Em computação, *plugin* é um programa usado para adicionar funções a outro programa maior, provendo alguma funcionalidade especial ou específica

[7] https://www.stanford.edu/

[8] http://swsa.semanticweb.org/content/swsa-ten-year-award-2014

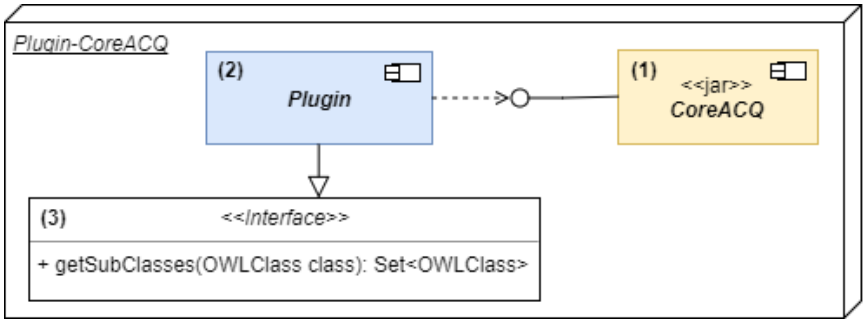

Figura 34 – Diagrama de componentes do *Plugin-CoreACQ*

de inferência para validar a consistência e deduzir novas informações de uma ontologia. Suas principais funcionalidades são:

- Edição de ontologia com suporte completo a OWL 2 DL.
- Suporte a motores de inferência, como *Pellet* e *Hermit*.
- Suporte a modularização de ontologias.
- Customização de componentes gráficos, como campos e botões, para que o usuário possa ocultar, exibir ou mudar suas posições.

4.3.2 Plugin-CoreACQ

O *Protégé* é capaz de integrar *plugins* em sua arquitetura, isto é, incorporar novas funcionalidades sem a necessidade de modificação em sua especificação. Para exemplificar, suas funções de raciocínio automático são proporcionadas e executadas por intermédio de *plugins* relacionados a motores de inferência (e.g. *Pellet*). A criação de um *plugin* requer, basicamente, que seu projeto siga algumas instruções e seja desenvolvido em Java, com seu executável disponibilizado junto aos arquivos de instalação do *Protégé*.

Em razão do suporte a *plugins*, é viável considerar o desenvolvimento de um *plugin* do *CoreACQ* para o sistema *Protégé*. Posto isto, a visão geral da arquitetura do *Plugin-CoreACQ* é apresentada na Figura 34. O componente (1) refere-se ao *framework CoreACQ* que foi construído e discutido ao longo desta dissertação (Vide Capítulo 3). O segundo (2) diz respeito ao *plugin* proposto nesta seção, cuja versão inicial propõe-se a consultar relacionamento de subsunção entre classes (e.g. $Woman \sqsubseteq Human$) por meio do componente (1). Já a interface pública (3) possibilita ao *Protégé* comunicar-se com o componente (2) para inferir relacionamentos entre subclasses (e.g. *Woman* e *Man*) e classes (e.g. *Human*).

4.3.3 Execução e Resultados do *Plugin-CoreACQ*

Durante a inicialização, o *Protégé* realiza uma busca em seus arquivos e localiza o executável do *Plugin-CoreACQ*, que prontamente é integrado ao sistema. Neste momento, o

Protégé exibe sua tela (GUI) inicial contendo uma ontologia vazia, conforme a Figura 35, a seguir.

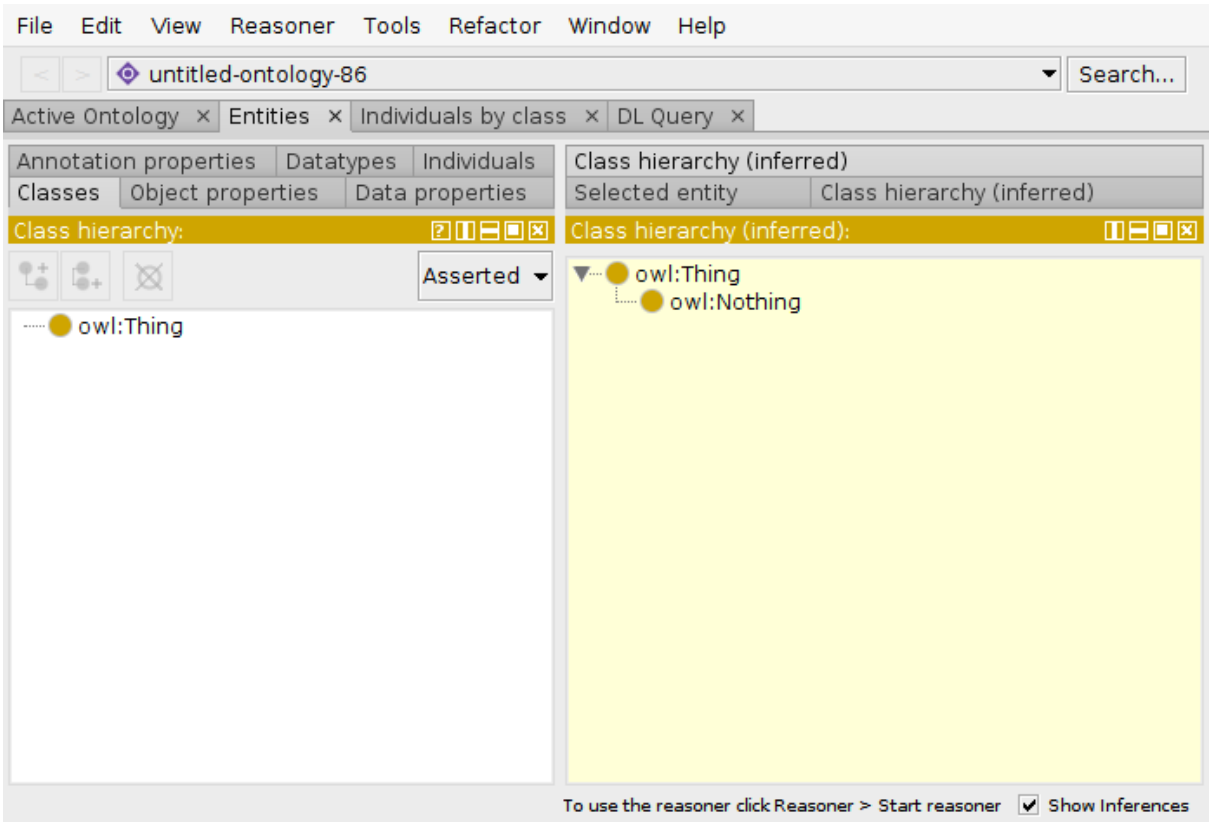

Figura 35 – Tela inicial do sistema *Protégé*

Após o *Protégé* ser inicializado, vem a ser possível a definição de classes na ontologia. Dessa forma, as seguintes classes foram definidas: *Animal* (Animal), *Hominid* (Hominídeo), *Human* (Humano), *Mammal* (Mamífero), *Man* (Homem), *Organism* (Organismo), *Primate* (Primata) e *Vertebrate* (Vertebrado). O resultado desse processo pode ser observado na Figura 36. Até então, nenhuma informação sobre a hierarquia de classes - isto é, relacionamentos entre subclasses e classes - foi considerada.

A partir de agora, a ontologia de exemplo possui a declaração de classes (Vide Figura 36). Portanto, o *Plugin-CoreACQ* pode ser ativado para determinar a hierarquia de classes por raciocínio automático de subsunção sobre a SUMO. O *plugin* de que se fala encontra-se disponível na aba de motores de inferência (*Reasoner*), como visto na Figura 37.

Depois do *Plugin-CoreACQ* ser ativado, o *Protégé* comunica-se com ele com a finalidade de adquirir conhecimentos a respeito de relacionamentos entre subclasses e classes. Para isso, o *plugin* consulta todas as classes da ontologia criada no *Protégé* e, em seguida, produz pares de classes, de acordo com os exemplos a seguir:

1. *Animal* e *Organism*

2. *Vertebrate* e *Animal*
3. *Mammal* e *Vertebrate*
4. *Primate* e *Mammal*
5. *Hominid* e *Primate*
6. *Human* e *Hominid*
7. *Man* e *Human*
8. *Woman* e *Human*

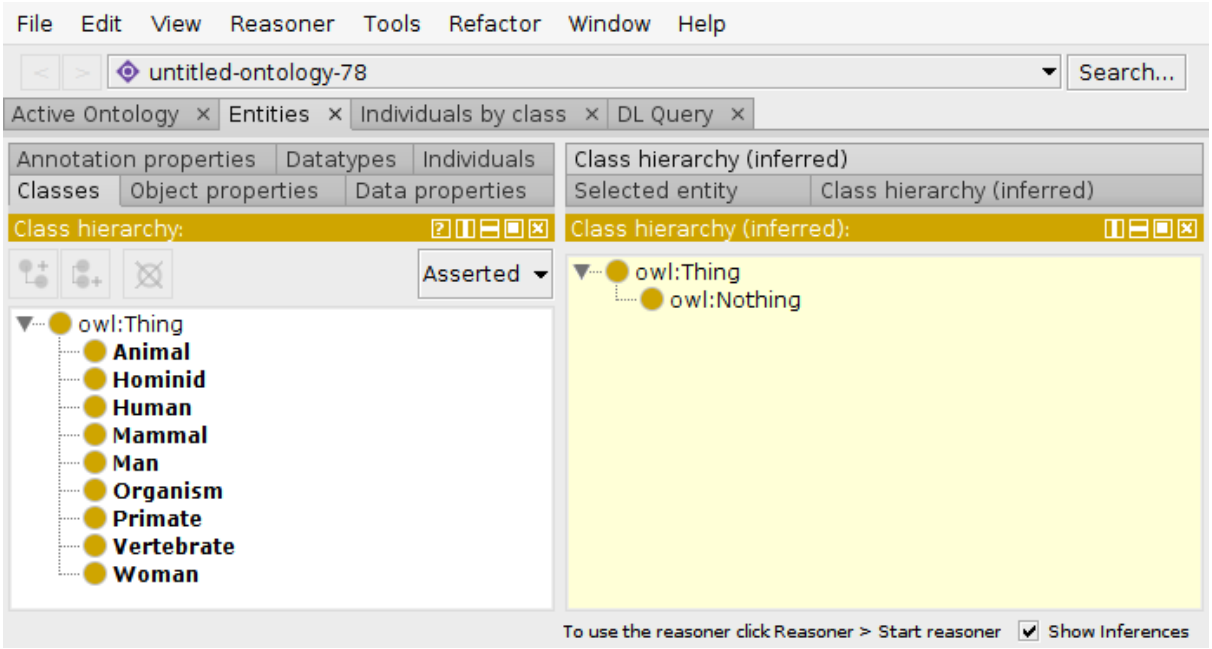

Figura 36 – Tela do sistema *Protégé* após a inclusão de novas classes na ontologia

Neste ponto, *Plugin-CoreACQ* envia cada par de classes (e.g. *Man* e *Human*) para o *CoreACQ* tentar provar se existe um relacionamento de subsunção (e.g. $Man \sqsubseteq Human$). Todos os pares cuja subsunção é comprovada são selecionados e enviados como resposta para o *Protégé*, que torna-se capaz de construir a hierarquia de classes automaticamente, como verificado na Figura 38.

O *Protégé* possui uma funcionalidade que permite exportar, em código OWL 2 DL, a hierarquia de classes construída por intermédio do *CoreACQ*. Essa funcionalidade fica disponível no menu *File* (Arquivo), conforme visto na Figura 39. Assim, um engenheiro de ontologias pode fazer uso das informações inferidas pela SUMO, que pode contribuir com a evolução de sua ontologia de domínio.

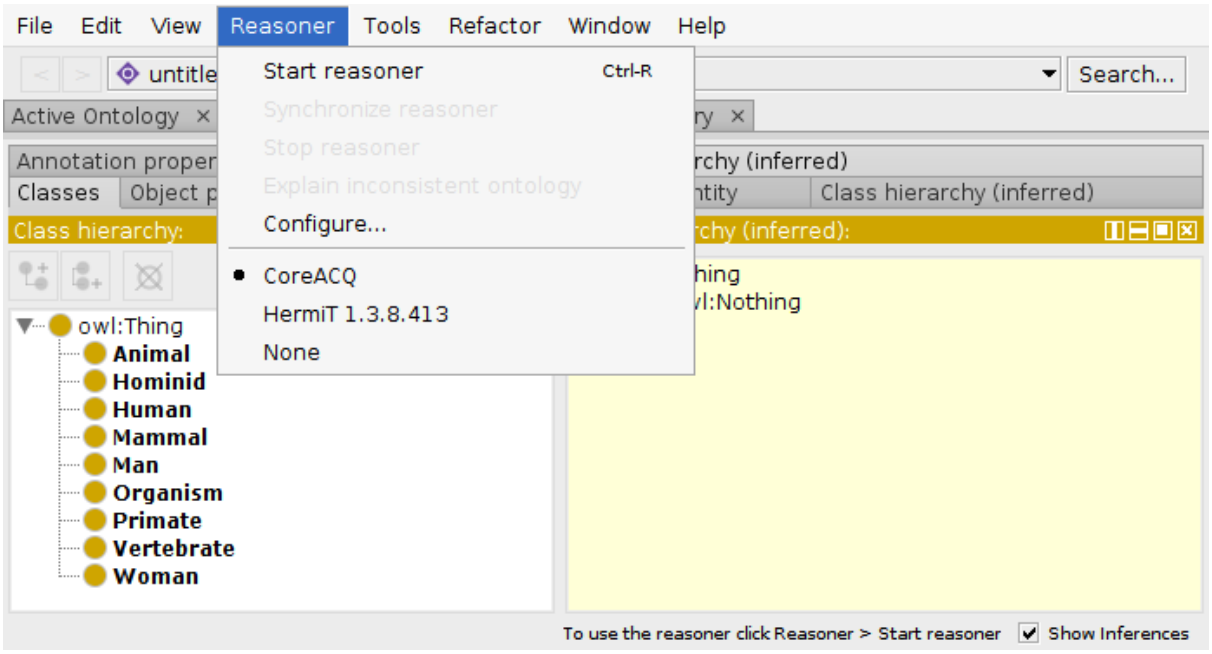

Figura 37 – Menu do sistema *Protégé* com a opção de inicializar o *CoreACQ-Plugin*

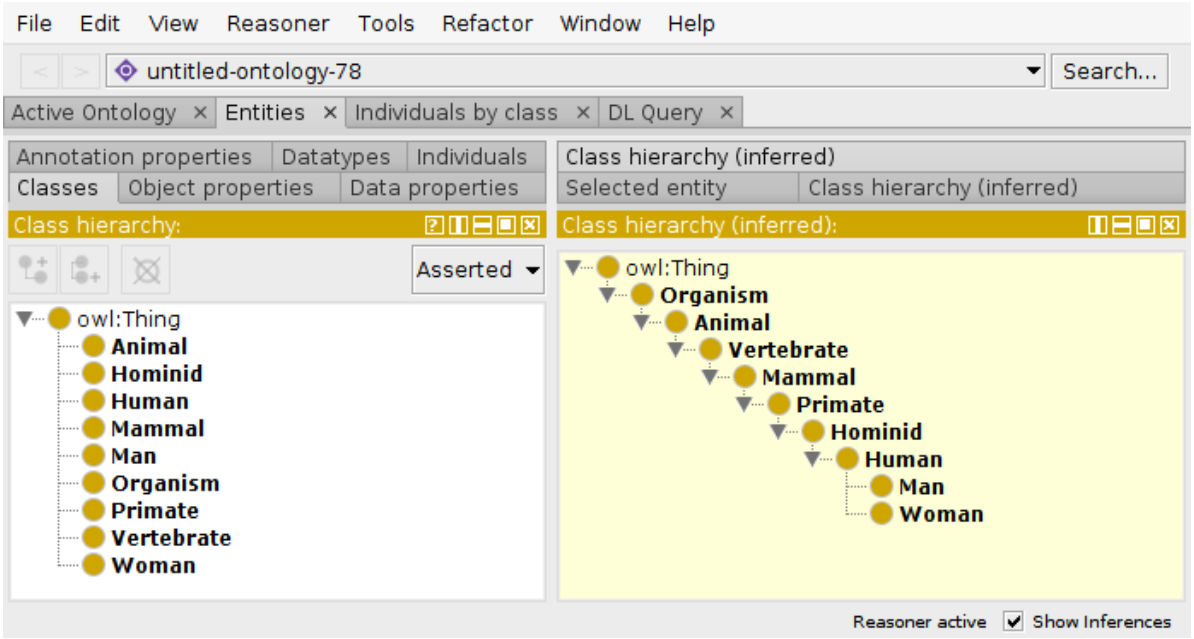

Figura 38 – Sistema *Protégé* exibindo a hierarquia de classes inferida por meio do *CoreACQ*

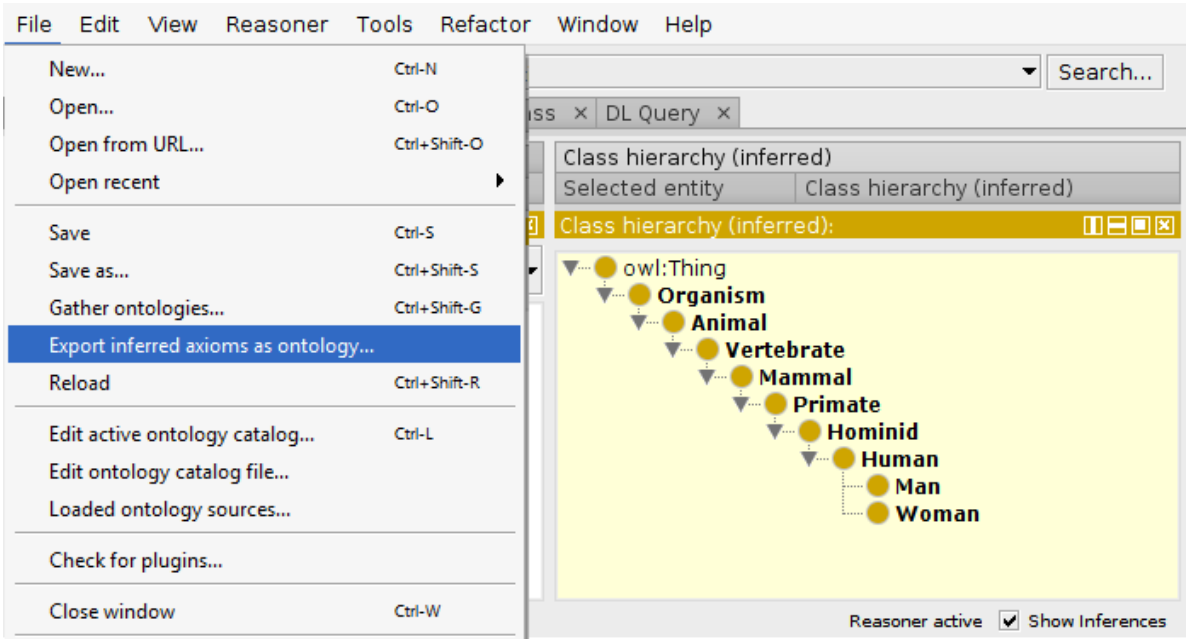

Figura 39 – Menu do Sistema *Protégé* com a função de exportar o código OWL 2 DL com a hierarquia de classes inferida por meio do *CoreACQ*

4.4 Conclusão

Este capítulo apresentou as etapas de definição, planejamento, análise, interpretação e avaliação no que se refere à qualidade do *CoreACQ*. Várias configurações distintas foram investigadas com o propósito de determinar o correto funcionamento do *framework*.

Cinco experimentos foram realizados considerando-se oito configurações do *CoreACQ*. Os resultados obtidos nesta fase permitiu avaliar se o *framework* estava ápto a realizar validações de CQs de forma precisa e em um tempo conveniente.

A hipótese nula de cobertura ($\mathbf{H}_{n1}$) foi rejeitada pelas configurações que empregam a *WordNet* para identificar conceitos desconhecidos pela *SUMO*, assim todas as CQs verdadeiras foram validadas nos experimentos. A hipótese nula de precisão ($\mathbf{H}_{n2}$) foi rejeitada considerando todas as configurações do *framework*, ou seja, nenhuma CQ falsa foi validada durante os experimentos. A hipótese nula de tempo de resposta ($\mathbf{H}_{n2}$) foi rejeitada pelas configurações que utilizam a função de seleção de premissas, dessa maneira foi possível validar cada CQ verdadeira em até 10 segundos. Todavia, declarou-se como desvantagem a probabilidade de determinadas CQs verdadeiras não serem validadas por causa da redução da base de conhecimento (Vide Seção 4.2). Ainda assim, foi evidenciado que o *CoreACQ* pode ser utilizado por ferramentas da engenharia de ontologias (e.g. *Protégé*) para proporcionar a evolução autônoma de uma ontologia de domínio.

No próximo capítulo, serão abordados os trabalhos relacionados a esta dissertação.

5 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados e comparados os trabalhos relacionados com objetivos e resultados semelhantes ao mesmo tema do nosso *framework*.

5.1 SigmaKEE

SigmaKEE (Sigma *Knowledge Engineering Environment*, em tradução livre Ambiente de Desenvolvimento para Engenharia do Conhecimento Sigma) é um sistema Java e JSP[1] de código aberto para desenvolvimento, visualização e teste de teorias em FOL. Ele é capaz de manusear conhecimentos em KIF e, ainda, possui otimizações para trabalhar com a SUMO (PEASE, 2003). Tratando-se de inferência tem o mesmo objetivo do CoreACQ, exercer raciocínio por intermédio de um sistema ATP para responder perguntas sobre uma base de conhecimento. Segundo Pease e Schulz (2014), ele consiste de um conjunto de ferramentas integradas no nível de usuário por uma interface HTML (*HyperText Marckup Language*, Linguagem de Marcação de Hipertexto), abrangendo:

- Tradução de teorias entre formatos THF (*Typed Higher-Order Form*, em tradução livre Fórmula de Alta Ordem), TPTP, SUO-KIF, OWL e *Prolog*.
- Mapeamento de teorias para outras teorias baseadas em similaridade de nomes de termos e definições.
- Navegação estruturada[2] de conteúdo de teorias ligadas e ordenadas por hiperlinks[3], incluindo apresentação hierárquica em árvore.
- Paráfrases em linguagem natural de teorias em diferentes linguagens.
- Navegação estruturada da *WordNet* e seus linlks para a SUMO.
- Vários tipos de ferramentas de análise estática para teorias, bem como inferências por um sistema ATP, que objetiva encontrar contradições teóricas.

Em relação ao componente de dedução, é apresentado na Figura 40 uma visão geral da integração entre *E Prover* - o mesmo sistema ATP usado nesta dissertação - e *SigmaKEE*. Inicialmente, ocorre a tradução da ontologia (e.g. SUMO) a ser utilizada para o formato TPTP. Em seguida, o *E Prover* é inicializado em modo interativo por meio de seu próprio executável, denominado *e_ltb_runner*, que recebe a ontologia em TPTP como argumento.

[1] http://www.oracle.com/technetwork/java/javaee/jsp/index.html

[2] Navegação estruturada é uma técnica usada em interfaces gráficas de usuário para fornecer um meio para localização de recursos dentro da estrutura de programas, documentos ou em páginas web

[3] Hiperlink ou link é um mecanismo de redirecionamento na Web, permite ao usuário acessar uma página HTML através de outra

Outros conhecimentos podem ser informados pelo usuário em uma sessão de trabalho separada, em outros termos, é possível definir axiomas temporários que são considerados apenas em uma determinada consulta. Quando o usuário deseja realizar uma inferência, *SigmaKEE* traduz a consulta e os axiomas da sessão corrente para a sintaxe TPTP e envia-os para o *e_ltb_runner*, que por sua vez tenta encontrar uma resposta. Em caso de sucesso, a prova encontrada é traduzida de volta para SUO-KIF e apresentada como resposta.

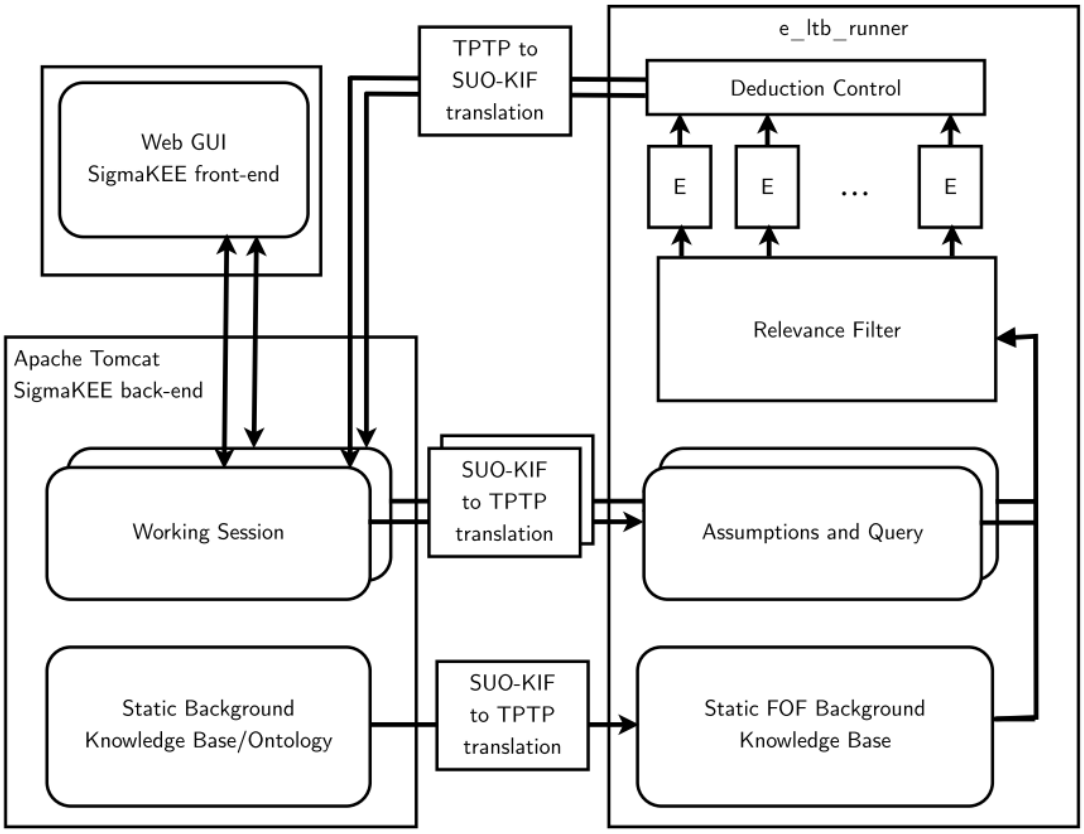

Figura 40 – Principais componentes arquitetônicos e fluxos de dados da SigmaKEE em relação ao componente de dedução

Fonte: Pease e Schulz (2014)

O componente de tradução de conhecimentos entre vários formatos mostra-se um recurso relevante do *SigmaKEE*. A conversão de axiomas entre as linguagens SUO-KIF e TPTP (FOL) não pode ser feita de modo direto, necessitando de várias transformações não triviais. A justificativa de tal complexidade diz respeito à menor expressividade de FOL em comparação com SUO-KIF. Portanto, ao contrário do *CoreACQ*, esse sistema está preparado para manipular a versão original da ontologia SUMO.

Devido a isso, como o próprio nome sugere, *SigmaKEE* é um ambiente de desenvolvimento voltado a engenheiros de ontologias e não pretende oferecer soluções específicas para programas de terceiros fazerem uso de suas funcionalidades. Um engenheiro de *soft-*

ware[4] deve realizar um estudo profundo no código-fonte[5] desse sistema antes de ser capaz de adaptá-lo para o seu programa. Essa é a principal diferença entre o *CoreACQ* e o *SigmaKEE*.

Tabela 11 – Tabela do comparativo entre as funcionalidades do *SigmaKEE* e do *CoreACQ*

Sistema	**ATP**	**Tradução entre**	**GUI**	**WordNet API**	**Mecanismo de Cache**	**Seleção de Premissas**
SigmaKEE	*E Prover* *Vampire*	SUO-KIF TPTP THF OWL Prolog	Web	Suporta	-	-
CoreACQ	*E Prover*	-	-	Suporta	Implementa	Implementa

Na Tabela 11 apresentamos uma comparação entre o *SigmaKEE* e o *CoreACQ*. O *SigmaKEE* suporta dois sistemas ATP e possui funções para tradução de teorias entre vários formatos, também oferece uma interface gráfica (GUI) por meio de uma plataforma *Web* e se comunica com a *WordNet* para consultar conceitos na SUMO. Em contrapartida, o *CoreACQ* implementa funcionalidades como mecanismo de *cache* para tentar agilizar a validação de CQs, seleção de premissas para reduzir o tempo do processo de raciocínio em um sistema ATP e, ainda, utiliza a *WordNet* para identificar conceitos em consultas em FOL.

5.2 Multiple ANSwer EXtraction

O sistema *Multiple ANSwer EXtraction* (*MANSEX*, em tradução livre Extração de Múltiplas Respostas) é um *framework* para extrair múltiplas respostas sobre uma mesma pergunta, interpretando-a como uma consulta em FOL com variáveis existencialmente quantificadas. De acordo com Sutcliffe, Yerikalapudi e Trac (2009), a disponibilidade limitada de respostas de perguntas em sistemas ATP da época foi o que motivou o desenvolvimento do seu *framework*. A contribuição do *MANSEX* foi melhorar o uso de um sistema ATP em virtude de tornar viável a aquisição de múltiplas respostas para uma pergunta. Sua estrutura básica foi implementada na linguagem Java para ser usada em conjunto com o projeto *SigmaKEE* (Vide Subseção 5.1). Dessa maneira, vem a ser capaz de interagir com a ontologia SUMO.

A abordagem adotada pelo *MANSEX* é enviar uma certa pergunta repetidas vezes para um sistema de base - isto é, um ATP - que possa provar e retornar respostas definitivas.

[4] Em ciência da computação, engenheiro de *software* é um profissional especializado no desenvolvimento e no aperfeiçoamento de programas de computador.

[5] Código-fonte refere-se as linhas de programação que formam um sistema de computador em sua forma original

Em cada iteração, a pergunta é expandida para negar as respostas já conhecidas, de modo que múltiplos resultados sejam extraídos. Para exemplificar, a seguir, são apresentadas uma lista de axiomas (5.1, 5.2, 5.3, 5.4, 5.5, e 5.6) e uma pergunta (5.7):

$$p(A, B) \tag{5.1}$$
$$p(A, f(B, C)) \tag{5.2}$$
$$\forall z \quad p(B,\, z) \tag{5.3}$$
$$\forall z \quad p(z,\, C) \tag{5.4}$$
$$A \neq B \tag{5.5}$$
$$A = C \tag{5.6}$$
$$\exists x, y \quad p(x,\, y) \tag{5.7}$$

Após o processamento em um sistema ATP, considera-se que a resposta para a pergunta (5.7) seja *p(A, B)*. Tendo em vista a obtenção de outro resultado, uma nova pergunta (5.8) é construída através da expansão de (5.7), conforme visto a seguir. Dessa forma uma segunda resposta pode ser obtida, por exemplo *p(A, C)*. Essa técnica iterativa continua enquanto diferentes respostas são adquiridas.

$$\exists x, y \quad p(x,\, y) \quad \wedge \quad (x \neq A \quad \vee \quad y \neq B)) \tag{5.8}$$

O principal componente do *MANSEX* é o *One Answer Extraction System* (OAESys, em tradução livre Sistema de Extração de Resposta Única), uma ferramenta capaz de processar uma resposta de um sistema ATP para uma pergunta (conjectura) com a finalidade de extrair os valores referentes a variáveis existencialmente quantificadas. Desse modo, a arquitetura de tal *framework* desempenha suas funcionaliades criando um canal de comunicação entre *OAESys* e um sistema ATP. Porém, a sua utilização está condicionada às funções e imposições do *SigmaKEE*, dificultando assim sua integração com ferramentas de terceiros.

Tabela 12 – Tabela do comparativo entre as funcionalidades do *MANSEX* e do *CoreACQ*

Sistema	**ATP**	**Múltipla Resposta**	**WordNet API**	**Mecanismo de Cache**	**Seleção de Premissas**
MANSEX	*E Prover* *Vampire*	Implementa	-	-	-
CoreACQ	*E Prover*	Suporta	Suporta	Implementa	Implementa

Apresentamos na Tabela 12 uma comparação entre o *MANSEX* e o *CoreACQ*. O *MANSEX* suporta dois sistemas ATP e possui uma implementação própria para proporcionar múltiplas respostas para uma mesma consulta, entretanto, a versão atual do *E*

Prover (Vide Seção 2.2.3 do Capítulo 2) já dispõe de uma função para múltiplas respostas. O *CoreACQ*, por sua vez, implementa um mecanismo de *cache* para tentar agilizar a validação de CQs, seleção de premissas para reduzir o tempo do processo de raciocínio em um sistema ATP e, também, comunicação com a *WordNet* para identificar conceitos em consultas em FOL.

5.3 Sistema CNL-WKR

O sistema *CNL-WKR* tem como objetivo responder consultas complexas por raciocínio automático em um sistema ATP, usando a linguagem natural controlada *Attempto Controlled English* (ACE, em tradução livre Inglês Controlado *Attempto*) para permitir a criação de perguntas, assim, a comunicação com seus usuários pode ser feita por intermédio de um idioma semelhante ao Inglês. Ao processar uma consulta, ele baseia-se em conhecimentos internos proporcionados pela SUMO e outras informações provenientes de fontes externas na *Web*. Seus principais pontos de interesse envolvem a tradução de consultas em Inglês para lógica, o alinhamento delas com as fontes externas - isto é, regularização de nomes de conceitos - e, além disso, conhecimentos fornecidos pela SUMO (DELLIS, 2010).

CNL-WKR utiliza o ACE, uma versão controlada do idioma Inglês mediante uma gramática que restringe os significados de todas as frases aceitáveis, com o intuito de evitar problemas com ambiguidades. Ela foi projetada como uma linguagem de especificação que combina a familiaridade da linguagem natural com o rigor das linguagens de especificação formal (FUCHS; SCHWERTEL; SCHWITTER, 1999).

De acordo com Dellis (2010), uma vantagem da ACE refere-se a existência de ferramentas capazes de converter tal linguagem em *Discource Representation Structure* (DRS, em tradução livre Estrutura de Representação de Discurso). DRS é um formato mais semelhante ao aceito por sistemas ATP devido a ser fundamentado em FOL, ou seja, busca facilitar a conversão de uma sentença ACE em um axioma TPTP. Para exemplificar, abaixo é mostrado uma frase ambígua e comumente usada entre humanos:

"Every man loves a woman"

Em ACE, a sentença acima é interpretada sem ambiquidades como a seguinte fórmula lógica:

$$\forall x \quad Man(x) \rightarrow \exists y \quad (Woman(y) \quad \wedge \quad loves(x, y))$$

Outra característica do *CNL-WKR* compreende o uso do *SPASS-XDB*, um sistema ATP que incorpora axiomas de conhecimento geral de múltiplas fontes externas durante o processo de dedução (SUTCLIFFE et al., 2010). Quando esse raciocinador é executado,

ele recebe um arquivo contendo informações sobre as fontes externas de conhecimento, os axiomas internos definidos pela SUMO e a consulta (ou conjectura) a ser respondida.

Apresentamos na Figura 41 uma visão geral do funcionamento do sistema *CNL-WKR*, sendo a primeira operação na parte inferior e a última na parte superior. Uma sentença em ACE é recebida como uma consulta de entrada e convertida em formato DRS, que por sua vez é transformado em um axioma TPTP. Nesse ponto ocorre o alinhamento de seus conceitos com as fontes externas e a SUMO. Em seguida, *SPASS-XDB* é inicializado para desempenhar seu processo de raciocínio e tentar responder a pergunta em questão.

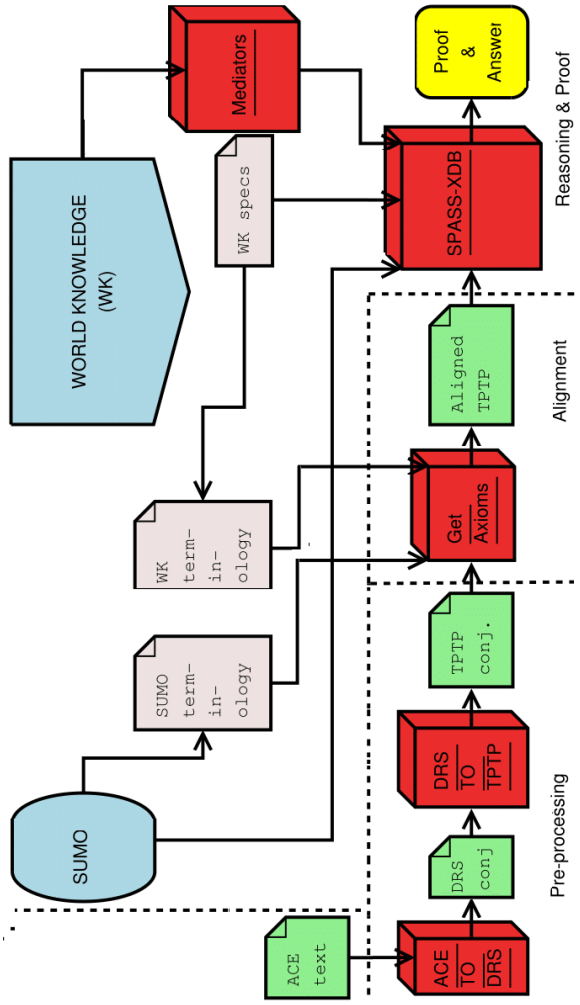

Figura 41 – Visão geral do sistema CNL-WKR

Fonte: Dellis (2010)

Na Tabela 13 apresentamos uma comparação entre o *CNL-WKR* e o *CoreACQ*. O *CNL-WKR* possui como diferencial uma função que interpreta consultas elaboradas em uma linguagem natural semelhante ao Inglês. No entanto, apenas o *CoreACQ* disponibiliza recursos como mecanismo de *cache* para agilizar a validação de CQs, seleção de premissas para reduzir o tempo do processo de raciocínio em um sistema ATP e, também, comunicação com a *WordNet* para identificar conceitos em consultas em FOL.

Tabela 13 – Tabela do comparativo entre as funcionalidades do *CNL-WKR* e do *CoreACQ*

Sistema	**ATP**	**Linguagem Natural**	**WordNet API**	**Mecanismo de Cache**	**Seleção de Premissas**
CNL-WKR	*SPASS-XDB*	Suporta	-	-	-
CoreACQ	*E Prover*	-	Suporta	Implementa	Implementa

5.4 Conclusão

Este capítulo teve como objetivo apresentar trabalhos relacionados ao mesmo tema desta dissertação. Neste capítulo, foram indicados os objetivos gerais e os resultados acerca de tais pesquisas e uma comparação com o nosso *framework*.

Na Tabela 14 apresentamos um resumo geral comparando todos os *frameworks* relacionados e o *CoreACQ*. Na primeira coluna são indicados os sistemas considerados. A segunda, terceira, quarta, quinta e sexta colunas apresentam os ATPs suportados, a capacidade de tradução de teorias, a disposição de interface gráfica (GUI), o tratamento de linguagem natural e a capacidade de fornecer múltiplas respostas levando em consideração cada *framework*. Já a sétima, oitava e nona colunas mostram quais deles implementam comunicação com a *WordNet*, mecanismo de *cache* e seleção de premissas.

Observam-se também as principais dissemelhanças entre os trabalhos relacionados. Em primeiro lugar, constata-se o uso de diferentes sistemas ATPs para realizar raciocínio automático sobre ontologias em FOL. Percebe-se que somente *SigmaKEE* oferece tradução de teorias entre diversos formatos e, também, uma interface gráfica por meio de uma plataforma *Web*. Em contrapartida, apenas *CNL-WKR* interpreta consultas elaboradas em uma linguagem natural semelhante ao Inglês. Já *MANSEX* possui uma implementação própria para proporcionar múltiplas respostas para uma mesma consulta, entretanto seus concorrentes - inclusive *CoreACQ* - utilizam sistemas ATPs que possibilitam resultados equivalentes.

Com base na sétima, oitava e nona colunas da Tabela 14, o *CoreACQ* detém três diferenciais se comparado aos demais trabalhos, são: (1) Comunicação com a *WordNet* para identificar conceitos não abrangidos pelo vocabulário da SUMO. *SigmaKEE* também faz uso de tal dicionário, todavia, apenas para consultar um determinado conceito, não para tratar consultas em FOL; (2) Mecanismo de *cache* para armazenar, em uma ontologia OWL 2 DL, consultas provadas de antemão por raciocínio sobre a SUMO. Assim, torna-se possível executar inferências para provar consultas futuras (e.g. CQs) evitando-se deduções em FOL (Vide Seção 3.1.4); e (3) Seleção de premissas para reduzir o tempo do processo de raciocínio em um sistema ATP, objetivando evitar axiomas não relevantes para a execução de uma consulta (Vide Seção 3.1.4).

Tabela 14 – Tabela do comparativo entre *CoreACQ* e os trabalhos relacionados

Sistema	**ATP**	**Tradução entre**	**Interface Gráfica**	**Linguagem Natural**	**Múltipla Resposta**	**WordNet API**	**Mecanismo de Cache**	**Seleção de Premissas**
SigmaKEE	*E Prover* *Vampire*	SUO-KIF TPTP THF OWL Prolog	Web	-	Suporta	Suporta	-	-
MANSEX	*E Prover* *Vampire*	-	-	-	Implementa	-	-	-
CNL-WKR	*SPASS-XDB*	-	-	Suporta	Suporta	-	-	-
CoreACQ	*E Prover*	-	-	-	Suporta	Suporta	Implementa	Implementa

6 CONCLUSÕES

A literatura indica que a *Web* Semântica e a sua principal camada, as ontologias, representam um recurso fundamental para que máquinas representem conhecimento e realizem raciocínio automático, o intuito é de prover máquinas capazes de entender o significado de documentos disponíveis na *Web*.

Artefatos, padrões e metodologias vêm sendo apresentados e desenvolvidos para apoiar o desenvolvimento de ontologias, como, por exemplo, avaliação de requisitos através de CQs. Quando uma CQ não é validada, significa que a ontologia não satisfaz os requisitos estabelecidos e, portanto, necessita da intervenção de seres humanos para resolver o problema. Assim, considera-se importante disponibilizar ferramentas com autonomia para buscar informações que minimizem processos manuais na validação de CQs, favorecendo, com isso, a avaliação de ontologias.

Este trabalho teve como finalidade o desenvolvimento do *CoreACQ*, um *framework* computacional para validação de CQs por raciocínio automático sobre a SUMO. Sua principal funcionalidade é a de inferência sobre axiomas em FOL por intermédio de um sistema ATP. Visando agilizar o processo de raciocínio, o *framework* possui funções como seleção de premissas por representação e busca de axiomas em grafos e, ainda, um mecanismo de *cache* por uma ontologia OWL 2 DL (Vide Seção 3.1.4 do Capítulo 3). Para tratar conceitos desconhecidos, também utiliza o dicionário léxico *WordNet* com o propósito de enriquecer o vocabulário da SUMO.

Os resultados dos experimentos realizados comprovaram a capacidade do *CoreACQ* em desempenhar suas funcionalidades. A seleção de premissas, por exemplo, possibilitou uma redução de, praticamente, 10 minutos no processo de validação de uma CQ. O mecanismo de *cache* proposto também foi capaz de reduzir o tempo de resposta do *framework*, porém atingiu um resultado menos significativo se comparado ao da função anterior. Além disso, mostrou-se como o dicionário *WordNet* pode tornar possível a identificação de conceitos não abrangidos diretamente pelo vocabulário da *SUMO*. Portanto, conclui-se que o *framework* proposto pode proporcionar autonomia para ferramentas que desejam evoluir ontologias de domínio evitando intervenção humana, uma vez que as hipóteses definidas não foram refutadas.

Este capítulo apresenta as conclusões acerca deste trabalho, evidenciando e indicando suas principais contribuições e propostas para trabalhos futuros.

6.1 Principais contribuições

As principais contribuições deste trabalho são descritas a seguir:

1. **Criação de um *framework* computacional para validar CQs**. A principal

contribuição deste trabalho é um *framework*, denominado *CoreACQ*, capaz de validar CQs por meio de informações deduzidas pela Ontologia de Topo SUMO, realizando raciocínio automático sobre ontologias em FOL por intermédio de um sistema ATP;

2. **Elaboração de uma arquitetura para manipular ontologias em FOL**. Outra contribuição deste trabalho foi a composição de uma arquitetura de referência, baseada em componentes, com o propósito de manipular ontologias e axiomas em FOL para a realização de inferências, assim, servindo de exemplo para novos projetos de arquiteturas com objetivos semelhantes;

3. **A função de seleção de premissas em ontologias em FOL**. Foi desenvolvido e apresentado no Capítulo 3, uma função para selecionar premissas por representação e busca de axiomas em grafos, buscando evitar a transmissão de axiomas não relevantes para um sistema ATP no processo de validação de uma CQ, o que contribui para uma redução no tempo de raciocínio.

4. **Incentivo ao reuso da Ontologia de Topo SUMO**. O reuso de informações semanticamente definidas por ontologias de topo livres e gratuitas, como a SUMO, é uma contribuição deste trabalho, posto que a definição formal de conceitos e documentos na *Web* não é uma tarefa trivial e exige a participação de um engenheiro de ontologias;

6.2 Trabalhos Futuros

Em Engenharia de *Software*, um *framework* preocupa-se em oferecer e aprimorar suas funcionalidades para melhor atingir seu objetivo, assim, encontram-se aspectos a serem aperfeiçoados no *CoreACQ*. Algumas sugestões para continuidade desta dissertação são mostradas a seguir:

1. **Realizar novos experimentos com a função de seleção de premissas**. Os experimentos realizados nesta dissertação avaliou a função de seleção de premissas com custos de caminho positivos e menores ou igual a cinco (Vide Capítulo 4). Assim, é importante considerar novos experimentos a fim de analisar o impacto e os resultados da seleção de premissas com custos de caminho maiores do que cinco.

2. **Realizar um estudo mais detalhado a respeito dos teoremas de Corretude e Completude**. São apresentadas na Seção 4.2, do Capítulo 4, informações sobre os teoremas de corretude e completude com relação ao *CoreACQ*. No entanto, um estudo mais detalhado torna-se importante, visto que a função de seleção de premissas causa impactos significativos no que se refere à propriedade de completude.

3. **Desenvolver um tradutor de KIF para FOL**. A versão original da SUMO lida com a linguagem KIF para representar seus axiomas. Não obstante, o processo de raciocínio do *CoreACQ* se baseia essencialmente em FOL. Por isso, permitir a conversão da SUMO em KIF para FOL faz-se um importante recurso para o *framework* apresentado.

4. **Desenvolver uma função de seleção de premissas baseada em um hipergrafo**. Hipergrafo é uma generalização de grafos onde é permitido que uma aresta relacione três ou mais nós. Dessa maneira, espera-se favorecer a representação e busca de axiomas por meio de um grafo.

5. **Implementar a comunicação com novos sistemas ATP**. Atualmente, somente o *E Prover* compõe o módulo responsável por realizar inferências sobre ontologias em FOL. Portanto, torna-se importante evitar que o *framework* proposto não dependa de um único sistema ATP para desempenhar suas atividades.

REFERÊNCIAS

ÁLVEZ, J.; LUCIO, P. **Adimen-SUMO**: Reengineering an Ontology for First-Order Reasoning. *International Journal on Semantic Web and Information Systems*, IGI Global, Hershey, PA, USA, v. 8, n. 4, p. 80–116, 2012. ISSN 1552-6283.

ÁLVEZ, J.; LUCIO, P.; RIGAU, G. **Evaluating the Competency of a First-Order Ontology**. In: *Proceedings of the 8th International Conference on Knowledge Capture*. New York, NY, USA: ACM, 2015. (K-CAP 2015), p. 28:1–28:4.

ARPÍREZ, J. C.; CORCHO, O.; FERNáNDEZ-LóPEZ, M.; GóMEZ-PÉREZ, A. **WebODE**: A Scalable Workbench for Ontological Engineering. In: *Proceedings of the 1st International Conference on Knowledge Capture*. New York, NY, USA: ACM, 2001. (K-CAP '01), p. 6–13. ISBN 1-58113-380-4.

BAADER, F.; HORROCKS, I.; SATTLER, U. **Description Logics**. In: HARMELEN, F.; LIFSCHITZ, V.; PORTER, B. (Ed.). *Handbook of Knowledge Representation*. [S.l.: s.n.], 2008. ISBN 9780444522115.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. **The Semantic Web**. [S.l.]: Sci. Am., 2001. 1-36 p. ISSN 0036-8733.

BEZERRA, C.; SANTANA, F.; FREITAS, F. **CQChecker**: A Tool to Check the Satisfaction of Description Logic Competency Questions on Ontologies. X Encontro Nacional de Inteligência Artificial e Computacional, 2013.

CARMEN, M.; PRADEL, C.; HERNANDEZ, N. **Verifying Ontology Requirements with SWIP**. In: *Proceedings of the 18th International Conference on Knowledge Engineering and Knowledge Management*. [S.l.: s.n.], 2012.

DELLIS, N. C. **Using Controlled Natural Language for World Knowledge Reasoning**. Dissertação (Mestrado) — University Of Miami, USA, 6 2010.

FERNÁNDEZ-LÓPEZ, M.; GÓMEZ-PÉREZ, A.; JURISTO, N. **Methontology**: From Ontological Art Towards Ontological Engineering. In: *Proceedings of the Ontological Engineering AAAI-97 Spring Symposium Series*. [S.l.]: American Asociation for Artificial Intelligence, 1997.

FILHO, D. M.; FREITAS, F.; OTTEN, J. **RACCOON**: A Connection Reasoner for the Description Logic ALC. In: EITER, T.; SANDS, D. (Ed.). *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. [S.l.]: EasyChair, 2017. (EPiC Series in Computing, v. 46), p. 200–211. ISSN 2398-7340.

FINLAYSON, M. A. **Java Libraries for Accessing the Princeton Wordnet**: Comparison and Evaluation. In: *Proceedings of the 7th Global Wordnet Conference*. [S.l.: s.n.], 2014.

FREITAS, F. **Ontologias e a Web Semântica**. In: VIEIRA, R.; OSóRIO, F. (Ed.). *Anais do XXIII Congresso da Sociedade Brasileira de Computação*. Campinas: [s.n.], 2003. v. 8, p. 1–52. ISSN 1981-6278.

FREITAS, F.; MALHEIROS, Y. **A Method to Develop Description Logic Ontologies Iteratively Based on Competency Questions: an Implementation**. In: *ONTOBRAS*. [S.l.: s.n.], 2013. v. 6, p. 142–153.

FREITAS, F.; OTTEN, J. **A Connection Calculus for the Description Logic ALC**. In: KHOURY, R.; DRUMMOND, C. (Ed.). *Advances in Artificial Intelligence*. Cham: Springer International Publishing, 2016. p. 243–256. ISBN 978-3-319-34111-8.

FUCHS, N. E.; SCHWERTEL, U.; SCHWITTER, R. **Attempto Controlled English**: Not Just Another Logic Specification Language. In: FLENER, P. (Ed.). *Logic-Based Program Synthesis and Transformation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 1–20. ISBN 978-3-540-48958-0.

GENNARI, J. H.; MUSEN, M. A.; FERGERSON, R. W.; GROSSO, W. E.; CRUBEZY, M.; ERIKSSON H. AND, N. N. F.; TU, S. W. **The evolution of Protégé**: an environment for knowledge-based systems development. *International Journal of Human-Computer Studies*, v. 58, n. 1, p. 89–123, 2003. ISSN 1071-5819.

HAASE, P.; LEWEN, H.; STUDER, R.; TRAN, D.; ERDMANN, M.; D'AQUIN, M.; MOTTA, E. **The NeOn Ontology Engineering Toolkit**. In: KORN, J. (Ed.). *WWW 2008 Developers Track*. [S.l.: s.n.], 2008.

KOVÁCS, L.; VORONKOV, A. **First-Order Theorem Proving and VAMPIRE**. In: SHARYGINA, N.; VEITH, H. (Ed.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 1–35. ISBN 978-3-642-39799-8.

MALHEIROS, Y.; FREITAS, F. **A Method to Develop Description Logic Ontologies Iteratively with Automatic Requirement Traceability**. In: *Informal Proceedings of the 27th International Workshop on Description Logics*. Vienna, Austria: [s.n.], 2014. p. 646–658.

MALHEIROS, Y.; FREITAS, F. **Unification in EL for competency question generation**. In: *Proceedings of the 30th International Workshop on Description Logics*. Montpellier, France: [s.n.], 2017. ISBN 1554-8929.

MCGUINNESS, D. L.; HARMELEN, F. **OWL Web Ontology Language Overview**. 2004. <https://www.w3.org/TR/owl-features/>. Acessado em 10 de Junho de 2018.

MINTO, S.; MURPHY, G. C. **Recommending Emergent Teams**. In: *Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: [s.n.], 2007. (MSR'07 ICSE Workshops), p. 5. ISSN 2160-1852.

MUSEN, M. A. **The Protégé Project**: A Look Back and a Look Forward. *AI Matters*, ACM, New York, NY, USA, v. 1, n. 4, p. 4–12, 2015. ISSN 2372-3483.

NILES, I.; PEASE, A. **Origins of The IEEE Standard Upper Ontology**. In: *Working Notes of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology*. Seattle: [s.n.], 2001. p. 37–42.

NILES, I.; PEASE, A. **Towards a Standard Upper Ontology**. In: *Proceedings of the International Conference on Formal Ontology in Information Systems*. New York, NY, USA: ACM, 2001. (FOIS '01, v. 2001), p. 2–9. ISBN 1-58113-377-4.

NOY, N. F.; MCGUINNESS, D. L. **Ontology Development 101**: A Guide to Creating Your First Ontology. *Stanford Knowledge Systems Laboratory*, 2001.

PARSIA, B.; SIRIN, E. **Pellet**: An OWL DL Reasoner. In: *International Workshop on Description Logics*. [S.l.: s.n.], 2004.

PEASE, A. **The Sigma Ontology Development Environment**. In: *Working Notes of the IJCAI-2003 Workshop on Ontology and Distributed System*. [S.l.: s.n.], 2003.

PEASE, A. **Standard Upper Ontology Knowledge Interchange Format Table of Contents**. 2004. <http://ontolog.cim3.net/file/resource/reference/SIGMA-kee/suo-kif.pdf>. Acessado em 22 de Junho de 2018.

PEASE, A.; SCHULZ, S. **Knowledge Engineering for Large Ontologies with SigmaKEE 3.0**. In: DEMRI, S.; KAPUR, D.; WEIDENBACH, C. (Ed.). *Automated Reasoning*. Cham: Springer International Publishing, 2014. v. 8562, p. 519–525. ISBN 978-3-319-08587-6.

PRESSMAN, R. S. **Engenharia de Software**. 5ª. ed. Rio de Janeiro: McGraw-Hill, 2002.

REN, Y.; PARVIZI, A.; MELLISH, C.; PAN, J. Z.; DEEMTER, K. van; STEVENS, R. **Towards Competency Question-Driven Ontology Authoring**. In: PRESUTTI, V.; D'AMATO, C.; GANDON, F.; D'AQUIN, M.; STAAB, S.; TORDAI, A. (Ed.). *The Semantic Web: Trends and Challenges*. Cham: Springer International Publishing, 2014. p. 752–767. ISBN 978-3-319-07443-6.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence**: A Modern Approach. 3ª. ed. New Jersey, USA: Prentice Hall, 2009.

SCHULZ, S. **System Description**: E.1.8. In: MCMILLAN, K.; MIDDELDORP, A.; VORONKOV, A. (Ed.). *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 735–743. ISBN 978-3-642-45221-5.

SHEARER, R.; MOTIK, B.; HORROCKS, I. **HermiT**: A highly-efficient owl reasoner. In: DOLBEAR, C.; RUTTENBERG, A.; SATTLER, U. (Ed.). *OWLED*. [S.l.]: CEUR-WS, 2008. (CEUR Workshop Proceedings, v. 432).

SOLINGEN, R.; BASILI, V.; CALDIERA, G.; ROMBACH, H. D. **Goal Question Metric (GQM) Approach**. In: *Encyclopedia of Software Engineering*. [S.l.: s.n.], 2002. ISBN 9780471377375.

STAAB, S.; STUDER, R.; SCHNURR, H.-P.; SURE, Y. **Knowledge processes and ontologies**. *IEEE Intelligent Systems*, v. 16, n. 1, p. 26–34, 2001. ISSN 1541-1672.

SURE, Y.; ERDMANN, M.; ANGELE, J.; STAAB, S.; STUDER, R.; ; WENKE, D. **Ontoedit**: Collaborative ontology development for the semantic web. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, UK, p. 221–235, 2002.

SUTCLIFFE, G.; SUDA, M.; TEYSSANDIER, A.; DELLIS, N.; De Melo, G. **Progress Towards Effective Automated Reasoning with World Knowledge**. In: *Proceedings of the 23nd International Florida Artificial Intelligence Research Society Conference*. Florida, USA: [s.n.], 2010. (FLAIRS-23), p. 110–115.

SUTCLIFFE, G.; YERIKALAPUDI, A.; TRAC, S. **Multiple Answer Extraction for Question Answering with Automated Theorem Proving Systems**. In: *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference*. Florida, USA: [s.n.], 2009. (FLAIRS-22), p. 105–110.

TAYLOR, N. J.; DENNIS, A. R.; CUMMINGS, J. W. **Situation Normality and the Shape of Search**: The Effects of Time Delays and Information Presentation on Search Behavior. *Journal of the American Society of Information Science and Technology*, v. 64, p. 909–928, 2013.

W3C OWL Working Group. **OWL 2 Web Ontology Language Document Overview**. 2012. <https://www.w3.org/TR/owl2-overview/>. Acessado em 27 de Maio de 2018.

WIVES, L. K. **Um estudo sobre agrupamento de documentos textuais em processamento de informações não estruturadas usando técnicas de "clustering"**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, Brasil, 1999.

WOHLIN, C.; RUNESON, P.; HOST, M.; OHLSSON, M.; REGNELL, B.; WESSLEN, A. **Experimentation in Software Engineering**. Berlin, Germany: Springer, 2012.

APÊNDICE A – INSTALAÇÃO E INICIALIZAÇÃO DO *COREACQ*

Neste apêndice são apresentadas informações e instruções para a instalação e configuração do *CoreACQ*, sendo todos os passos realizados através do sistema operacional *Ubuntu*[1]. O primeiro passo é realizar o *download*[2] dos arquivos do *E Prover*[3] disponível em seu site. Para facilitar o download, os arquivos estarão compactados[4] dentro de um único arquivo denominado *E.tgz*, que permite extrair seus arquivos em uma pasta denominada *E*, conforme visto na Figura 42.

Figura 42 – Arquivos para instalação do *E Prover ATP*

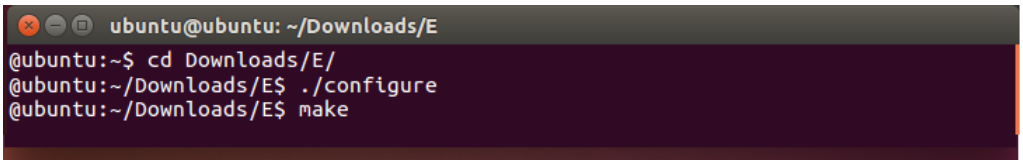

Figura 43 – Instalação do *E Prover ATP* por meio de um terminal do sistema operacional Ubuntu

O passo seguinte é a instalação do *E Prover* no sistema operacional *Ubuntu*, que deve ser executada por intermédio de um Terminal[5], como pode ser visto na Figura 43. Antes

[1] Ubuntu é um sistema operacional ou sistema operativo de código aberto, construído a partir do núcleo Linux

[2] Em redes de computadores, um *download* significa receber dados de um sistema remoto, como um site na *Web*

[3] https://wwwlehre.dhbw-stuttgart.de/ sschulz/E/Download.html

[4] Compactação de arquivos é um recurso usado para agrupar arquivos e reduzir o volume de tráfego em uma rede de computadores

[5] Em Ciência da COmputação, um terminal pode ser entendido como um programa que permite a execução de linhas de comando para manipular outro programa ou o sistema operacional

deve ser acessado o diretório com os arquivos do *E Prover*, para isso basta digitar "*cd Downloads/E*" no terminal. Após, é necessário executar o comando "*./configure*" e, em seguida, "*make*" para realizar a instalação. Como resultado, é criado um executável chamado *e_ltb_runner* dentro da pasta *Downloads/E/PROVER*, conforme visto na Figura 44, a seguir.

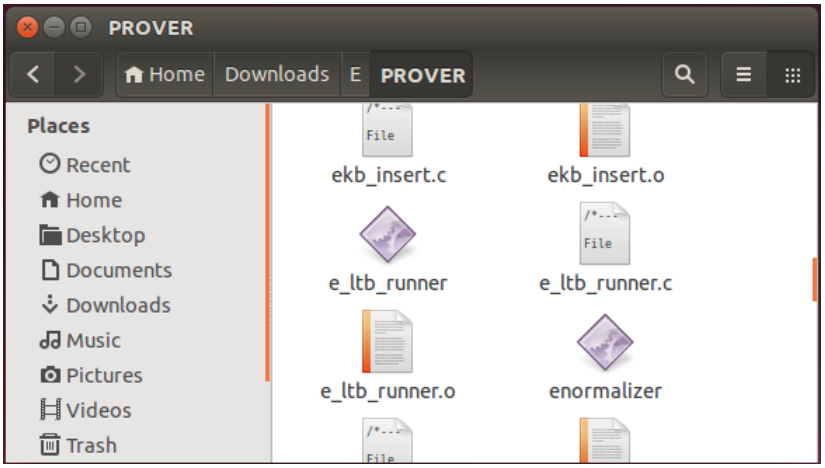

Figura 44 – Executável para inicializar o sistema *E Prover ATP*

O próximo passo é acessar os arquivos da *Adimen-SUMO v2.6*[6] disponibilizados por meio de seu site. Os arquivos também estarão compactados em um único arquivo denominado *AdimenSUMO2.6.zip*, que permite extrair seus arquivos em uma pasta denominada *AdimenSUMO2.6*, como visto na Figura 45, a seguir.

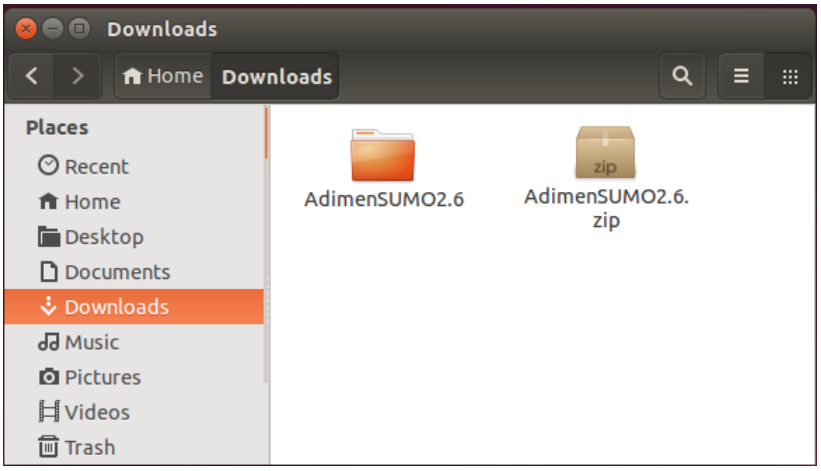

Figura 45 – Arquivos da Ontologia de Topo *Adimen-SUMO v2.6*

Na hipótese do componente *WordNet*[7] ser habilitado, os seus arquivos precisam ser baixados e disponibilizados no computador. Seu site possibilita o download do arquivo

[6] http://adimen.si.ehu.es/web/AdimenSUMO
[7] https://wordnet.princeton.edu/download/current-version

wn3.1.dict.tar.gz, que permite extrair seus arquivos em uma pasta denominada *dict*, conforme a Figura 46.

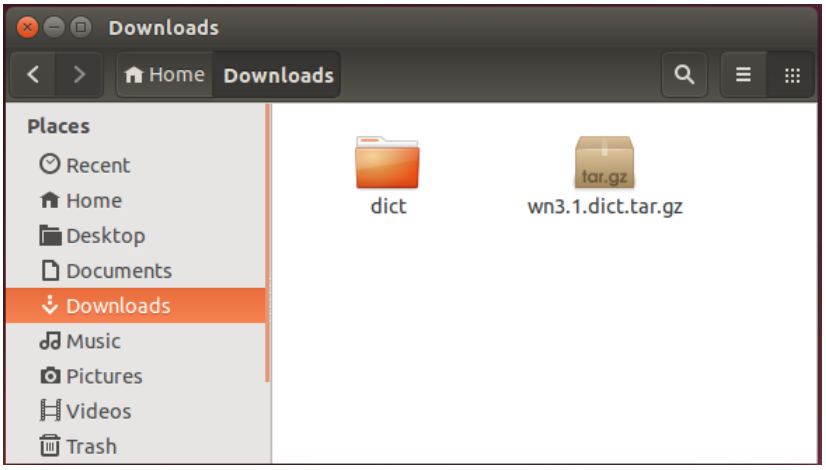

Figura 46 – Arquivos do dicionário léxico *WordNet*

O projeto do *CoreACQ* foi desenvolvido na linguagem de programação Java, sendo gerado um arquivo de extensão "jar" que possibilita suas funcionalidades dentro de outros sistemas Java, como visto na Figura 47. Assim, um sistema externo - isto é, de terceiros - deve adicionar o arquivo *coreacq-1.0.0.jar* em sua arquitetura para executar as funções do *CoreACQ*.

Figura 47 – Arquivo jar que deve ser importado no sistema externo para executar as funcionalidades do *CoreACQ*

Após a execução dos passos acima, um sistema externo pode configurar e inicializar o *CoreACQ*, como exemplificado no Código Java A.1, a seguir. A comunicação com o *E Prover* é configurada conforme a linha 10, onde são informados o caminho para o arquivo *e_ltb_runner* e o tempo máximo, em segundos, permitido para cada inferência. O módulo *SUMO Inference Engine* (Vide Seção 3.1.4 do Capítulo 3) é configurado através de dois parâmetros, como visto na linha 12, o primeiro recebe o objeto *eprover* criado na linha

10, já o segundo recebe o caminho para os arquivos da *WordNet*. O mecanismo de *cache* pode ser habilitado conforme a linha 13, deve ser informado o diretório onde a ontologia *cache* vai ser armazenada e, ainda, qual motor de inferência OWL 2 DL vai ser utilizado. Para habilitar a função de seleção de premissas é necessário informar o custo máximo de caminho, como visto na linha 14, que será usado pelo algoritmo de representação e busca de axiomas em grafos (Vide Seção 3.1.3 do Capítulo 3). Por fim, o *CoreACQ* é inicializado com o caminho para o arquivo da ontologia *Adimen-SUMO v2.6*, conforme a linha 15. A partir de agora, todos os métodos implementados para validar CQs estão prontos para serem executados, como, por exemplo, consultar se a classe *Woman* (Mulher) é subclasse de *Human* (Humano), conforme visto na linha 17.

Código A.1 – Exemplo em Java de como um sistema externo deve configurar o *CoreACQ*

```
1   public class SistemaExterno {
      public static void main(String[] args) {
3         String kbFilename = "Downloads/AdimenSUMO2.6/E-KIFtoFOF/Axioms/adimen.sumo
              .eprover.tstp";
          String wordNetPath = "Downloads/wordnet31/dict";
5         String eproverPath = "Downloads/E/PROVER";
          String cachePath = "Downloads/OntologiaCache";
7         int timeout = 10;
          int custoMaximoCaminho = 1;
9
          IProver eprover = EProverFactory.buildLinuxEProver(eproverPath, timeout);
11
          ISUMOEngineSync engine = InferenceEngineFactory.buildSUMOEngineSync(
              wordNetPath, eprover);
13        engine.loadCacheOntology(cachePath, Reasoner.RACCOON);
          engine.setAxiomsLimitByQuery(custoMaximoCaminho);
15        engine.start(kbFilename);

17        IProof result = engine.querySubclass("Woman", "Human");
      }
19  }
```