UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

HUGO ELIAS MARREDO CONSTANTINOPOLOS

Considerando aspectos éticos na construção de robôs

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Renan Maffei

Porto Alegre
2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Prof[a]. Patricia Pranke
Pró-Reitora de Graduação: Prof.[a] Cíntia Inês Boll
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Cláudio Machado Diniz
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

" Beware; for I am fearless, and therefore powerful.

" — MARY SHELLEY, FRANKENSTEIN

AGRADECIMENTOS

Agradeço a todos os amigos especiais que conheci na UFRGS, a todos os maravilhosos professores que me mostraram um mundo muito maior, a todos os momentos especiais que vivi nessa instituição: Aulas, trocas com colegas, viagens, festas, intercâmbio, estágios. Obrigado a tudo e todos que tornaram esse momento possível.

RESUMO

A ampla utilização da tecnologia, sobretudo da robótica e das técnicas de aprendizado de máquina e de inteligência artificial, no cotidiano de grande parte da população gera uma grande preocupação sobre qual a melhor maneira de abordar a ética e a moral das decisões tomadas por sistemas computacionais. Este estudo parte do padrão internacional da IEEE *Ontological Standard for Ethically Driven Robotics and Automation Systems*, onde é proposto a definição de um padrão ontológico para estabelecer uma metodologia para o projeto de sistemas robóticos e de automação que consideram aspectos éticos. Este estudo propõe uma das primeiras implementações de um aplicativo que explora os conceitos apresentados na ontologia proposta pelo padrão. É apresentado uma ferramenta que implementa uma parte do padrão ontológico, a de gestão de violações éticas, e fornece mecanismos para construir cenários dentro da ontologia, simular tais cenários, realizar a análise de eventos sobre essa ontologia, e visualizar os relacionamentos das entidades que fazem parte de um cenário construído na ferramenta. Como resultado desse estudo temos não só a construção de uma ferramenta que usa o padrão ontológico para sistemas robóticos e de automação que consideram aspectos éticos, mas também resultados gerados pelo uso da ferramenta. Com este trabalho é possível ter, também, mais dados para avaliar se a utilização do padrão ontológico apresenta resultados positivos para a construção de sistemas reais, qual a complexidade envolvida em utilizar o modelo, e como sistemas robóticos e de automação podem melhor utilizar o modelo.

Palavras-chave: Robótica. Ética. Ontologia.

Considering Ethical Aspects in the Construction of Robots.

ABSTRACT

The wide utilization of technology, specially for robotics and machine learning and artificial intelligence techniques for big part of the population brings a major concern on how is the proper way to approach ethics and moral regarding decisions that were taken by computational systems. This study starts from the results from another study: *Ontological Standard for Ethically Driven Robotics and Automation Systems*, which proposes the definition of a ontological standard to establish a methodology to design ethically driven robotics and automation systems. This present study proposes, thus, one of the very first implementation of an application that explores the concepts introduced on this ontological standard. The proposal is an application that implements a subset of the ontological pattern, the ethical violation management subsystem, providing mechanisms so that it is possible to built scenarios on top of the ontological standard, simulate such scenario, perform the analysis of all events that happens on such scenario and visualize the relationship between all entities that are part of the scenario built using the application. More over, the results of this study is not only the development of this application, which is one of the first to consider the ontological standard for ethically driven robotics and automation systems, and the data that were generated by the analysis performed by this tool, but also the data produced by utilizing an application built on top of the ontology for the very first time. It is now possible to better evaluate if the utilization of this ontological standard fits the needs and positively affects the development of real world applications, what is the complexity involved to properly utilize the model, and how robotics and automation systems can better utilize the model.

Keywords: Robotics, Ethics, Ontology.

LISTA DE ABREVIATURAS E SIGLAS

GVE	Gestão de violações éticas
IA	Inteligência artificial
NPE	Normas e princípios éticos
OAN	Ontologia de alto nível
OWL	Web Ontology Language
POO	Programação Orientada a Ontologias
PPD	Privacidade e proteção de dados
TPC	Transparência e prestação de contas

LISTA DE FIGURAS

Figura 2.1 Exemplo de uma ontologia para definir o conhecimento sobre pizza 17
Figura 2.2 Modelo parcial da ontologia NPE 21
Figura 2.3 Modelo parcial da ontologia GVE 22

Figura 4.1 Relacionamentos entre as classes *Ethical Behavior Monitor* e *Norm Violation* 35
Figura 4.2 Cenário de exemplo da utilização do comando de busca 46
Figura 4.3 Cenário de exemplo da utilização do comando de busca com indivíduos não conectados 48
Figura 4.4 Cenário de exemplo da utilização do comando de busca com múltiplos indivíduos conectados 49

Figura 5.1 Representação da modelagem do cenário de atribuição de responsabilidade ética e legal 52
Figura 5.2 Representação do cenário após o *reasoning* do cenário 53
Figura 5.3 Representação do cenário 2 58
Figura 5.4 Representação do cenário 2 após a percepção do movimento do robô 61

LISTA DE TABELAS

Tabela 2.1 Classes da ontologia **GVE**..23

Tabela 4.1 Indivíduos de uma ontologia de gestão de violações éticas 36
Tabela 4.2 Relacionamento entre indivíduos de uma ontologia de gestão de violações éticas 37

SUMÁRIO

1 INTRODUÇÃO **12**
1.1 Motivação e escopo **13**
1.2 Objetivos **14**
1.3 Organização deste documento **15**
2 FUNDAMENTAÇÃO TEÓRICA E REVISÃO DA LITERATURA **16**
2.1 Ontologia **16**
2.2 Robótica e Ética **18**
2.3 Primeiro padrão ontológico global para desenvolvimento de robôs e sistemas de automação que consideram aspectos éticos **20**
2.3.1 Ontologia para normas e princípios éticos 20
2.3.2 Ontologia para privacidade e proteção de dados 21
2.3.3 Ontologia para transparência e prestação de contas 21
2.3.4 Ontologia para gestão de violações éticas 22
2.3.5 Possíveis casos de uso do padrão ontológico 23
2.4 Programação orientada a ontologias **24**
2.4.1 OWL 25
2.4.2 Owlready2 26
3 METODOLOGIA **27**
3.1 Sistema desenvolvido **27**
3.1.1 Tecnologias Utilizadas 28
3.1.2 Aplicação da ferramenta desenvolvida 28
3.1.2.1 Criação de cenários que explorem a utilização dos conceitos éticos 28
3.1.2.2 Visualização dos resultados 29
4 IMPLEMENTAÇÃO **30**
4.1 Carregamento da ontologia OWL **31**
4.2 Acessando as classes da ontologia **32**
4.3 Acessando os indivíduos da ontologia **32**
4.4 Instanciando classes e criando indivíduos **33**
4.5 Definindo relacionamentos entre indivíduos **34**
4.6 Raciocinar sobre os dados da ontologia **35**
4.7 Carregamento de indivíduos e propriedades **36**
4.8 Visualização dos indivíduos e propriedades **38**
4.9 Navegação por linha de comando **41**
4.9.1 Comando *list* 41
4.9.2 Comando *select* 42
4.9.3 Comando *show* 43
4.9.4 Comando *follow* 44
4.9.5 Comando *search* 45
5 RESULTADOS **50**
5.1 Cenário 1 - Atribuição de responsabilidade ética ou legal **50**
5.1.1 Visualização do cenário 53
5.2 Cenário 2 - Violação ética, causa, justificativa, e plano de ação **55**
6 CONSIDERAÇÕES FINAIS **62**
REFERÊNCIAS **64**
APÊNDICE A — MANUAL DA FERRAMENTA **66**
A.1 Utilização do aplicativo **66**
A.2 Comandos para navegação por linha de comando **68**
A.2.1 Comando *help* 68

A.2.2 Comando *exit* 69
A.2.3 Comando *list* 70
A.2.4 Comando *select* 70
A.2.5 Comando *show* 72
A.2.6 Comando *unselect* 73
A.2.7 Comando *follow* 74
A.2.8 Comando *ifollow* 75
A.2.9 Comando *search* 77
APÊNDICE B — CLASSES E RELACIONAMENTOS DA ONTOLOGIA GVE.80

1 INTRODUÇÃO

Robôs são equipamentos mecânicos versáteis, equipados com atuadores e sensores, e sob o controle de um sistema computacional para executar tarefas motoras em ambientes físicos (HALPERIN; KAVRAKI; LATOMBE, 1999). Dentro da robótica, há soluções para as mais variadas atividades, desde automação industrial e carros autônomos até nano robôs que navegam dentro do corpo humano (LI et al., 2017). Uma das soluções empregada em várias situações é a navegação autônoma *indoor*, seja para carregar materiais dentro de uma linha de montagem, para realizar vigilância ou para realizar a limpeza e aspirar o pó de um ambiente. Nos últimos anos, temos observado uma enorme evolução nesses mecanismos, tanto na parte mecânica, com robôs de alta precisão para a realização de procedimentos médicos, ou aqueles que realizam trabalhos de extrema força e agilidade, como os presentes na indústria automotiva, quanto nos sistemas computacionais que controlam tais equipamentos. Com o aumento do desempenho, e diminuição da área, da potência consumida e do preço de microprocessadores e memórias, o hardware embarcado nesses sistemas é cada vez mais poderoso, permitindo que técnicas e algoritmos que até então eram executados apenas em poderosos computadores pudessem ser embarcados também. O que se observa, especialmente por causa da evolução do poder computacional dos sistemas embarcados, é que um grande número de soluções de automação utilizam-se de algoritmos de aprendizado de máquina.

Algoritmos de aprendizado de máquina são caracterizados por se treinarem automaticamente através de experiência e do uso de dados (MITCHELL, 1997). A utilização desses algoritmos, ao mesmo tempo que possibilita grandes avanços quanto as áreas de atuação de robôs autônomos, também cria um cenário em que o comportamento do robô é imprevisível, ou melhor, apesar de o objetivo do robô ser modelado por seus desenvolvedores, os critérios que o robô vai utilizar para atingir tal objetivo não é definido por seus projetistas, criando-se assim um cenário diferente do que tem-se até o momento, onde robôs seguiam programas determinísticos, programados de forma imperativa, e qualquer ação desse mecanismo fora prevista por seu criador.

Assim, em uma realidade onde robôs possuem comportamentos imprevisíveis, ao mesmo tempo que assumem papeis cada vez mais decisivos na vida humana, surge a discussão de como esses robôs irão se relacionar com a vida humana do ponto de vista de responsabilidades sociais, com questionamentos que antes estavam restritas à ficção científica, como: "De quem é a responsabilidade caso um robô cometa um crime?", ou

"Que tipo de informação um robô tem o direito de obter e guardar?", ou "Um robô pode omitir informações a um humano?". A ficção científica, durante décadas, vem abordando tais questionamentos, uma das obras literárias mais famosas a abordar esse tema é o conto Runaround, onde Isaac Asimov propõe as três leis da robótica (ASIMOV, 1942), são elas:

1. Um robô não pode ferir um ser humano ou, por inação, permitir que um ser humano sofra algum mal.
2. Um robô deve obedecer às ordens que lhe sejam dadas por seres humanos, exceto nos casos em que entrem em conflito com a Primeira Lei.
3. Um robô deve proteger sua própria existência, desde que tal proteção não entre em conflito com a Primeira ou Segunda Leis.

É correto afirmar que não estamos no ponto de que os robôs são seres conscientes, e que essas três regras são muito amplas para os problemas que enfrentamos, mas esse tipo de pensamento tem seu valor quando pensamos, como projetistas de robôs, se devemos ignorar a ética de um robô e segui-los tratando como mecanismos sob nosso total controle, ou se devemos nos antecipar e começar a prever qual o impacto de considerar algum aspecto ético na construção desses dispositivos que ano a ano estão se tornando mais fundamentais na nossa sociedade, e pensar em novas metodologias para a construção desses robôs que levem em conta essas preocupações.

Tendo em vista este panorama, um trabalho pioneiro em propor uma metodologia para o projeto dessa próxima geração de robôs é o primeiro padrão ontológico para robôs e sistemas autônomos que consideram aspectos éticos (PRESTES et al., 2021). Um padrão ontológico, ou baseado em ontologia, refere-se a uma estrutura conceitual que descreve e organiza o conhecimento sobre um domínio específico de maneira formal e semântica. Ela define as relações, propriedades e classes de objetos dentro desse domínio, permitindo uma representação clara e precisa das informações.

O presente estudo parte desse padrão ontológico para robôs e sistemas autônomos que consideram aspectos éticos e propõe uma das primeiras ferramentas para auxiliar na decisão de sistemas robóticos a utilizar a ontologia.

1.1 Motivação e escopo

A motivação deste trabalho reside na necessidade crescente de compreender e avaliar as ações e interações de robôs em diversos cenários, à medida que eles desempenham

papéis cada vez mais relevantes em nossa sociedade. Diante disso, o escopo deste projeto é desenvolver uma ferramenta para análise de eventos robóticos com base em um padrão ontológico estruturado. A ontologia servirá como uma fundação semântica para categorizar e interpretar eventos, permitindo uma análise mais profunda das ações dos robôs e de suas implicações éticas. O software resultante proporcionará *insights* valiosos para pesquisadores, engenheiros e especialistas em ética. Além disso, o trabalho busca estabelecer um marco inicial na integração prática de ontologias na análise de eventos robóticos, pavimentando o caminho para aplicações mais avançadas nesse campo em constante desenvolvimento.

1.2 Objetivos

O objetivo deste trabalho é explorar o desenvolvimento de sistemas de robótica ética por meio do uso de uma ontologia ética. Especificamente, o trabalho se concentrará no desenvolvimento de um aplicativo que utilize o padrão ontológico, pois se tratando de um modelo muito recente, o desenvolvimento de qualquer ferramenta que utilize-o como base gerará conhecimento para entender quais são os benefícios encontrados nessa metodologia e as dificuldades que outros engenheiros e projetistas de software e de sistemas robóticos enfrentarão.

Sobre o aplicativo a ser desenvolvido e apresentado nesse estudo, o objetivo é criar uma ferramenta que possibilite a simulação e análise de eventos robóticos que aconteçam dentro de um cenário modelado sobre a ontologia ética. A ferramenta englobará a modelagem de cenários de decisão ética, a visualização desses dados, a consulta por informações presentes no cenários modelados e a navegação por dentre os diversos relacionamentos desses indivíduos.

A principal pergunta que estamos tentando responder através do desenvolvimento desse trabalho é: como pode um padrão ontológico ser usado para desenvolver sistemas de robótica ética que estejam alinhados com os valores humanos e princípios éticos? Além disso, outra pergunta a ser respondida é como a ontologia pode ser usada para informar os processos de tomada de decisão de robôs autônomos em cenários do mundo real, e quais são os potenciais benefícios e limitações dessa abordagem.

1.3 Organização deste documento

Este trabalho está dividido da seguinte forma: o capítulo 2 fornece uma revisão da literatura relevante ao tema do trabalho. Serão apresentados os principais conceitos e teorias relacionadas ao assunto, além de trabalhos anteriores relevantes, destacando as contribuições e limitações. O capítulo 3 descreve a metodologia utilizada para a realização deste trabalho. São apresentados os procedimentos metodológicos adotados, incluindo uma apresentação da aplicação que será desenvolvida, as técnicas de coleta de dados, de análise e quais as metodologias adotadas para a interpretação dos resultados. O capítulo 4 apresenta em maiores detalhes o funcionamento da aplicação construída nesse trabalho. Abordando as tecnologias utilizadas, como as ferramentas utilizadas auxiliam na construção do aplicativo, e um manual de uso para o usuário dessa ferramenta. O capítulo 5 apresenta resultados da utilização do aplicativo. Aqui são propostos dois casos de uso do aplicativo que abordam diferentes conceitos do padrão ontológico, e que apresentam casos de uso para a ferramenta. Por fim, o capítulo 6 apresenta as conclusões do trabalho, incluindo respostas às questões iniciais quanto a utilização do padrão ontológico para auxiliar sistemas robóticos, as contribuições do trabalho, suas limitações e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA E REVISÃO DA LITERATURA

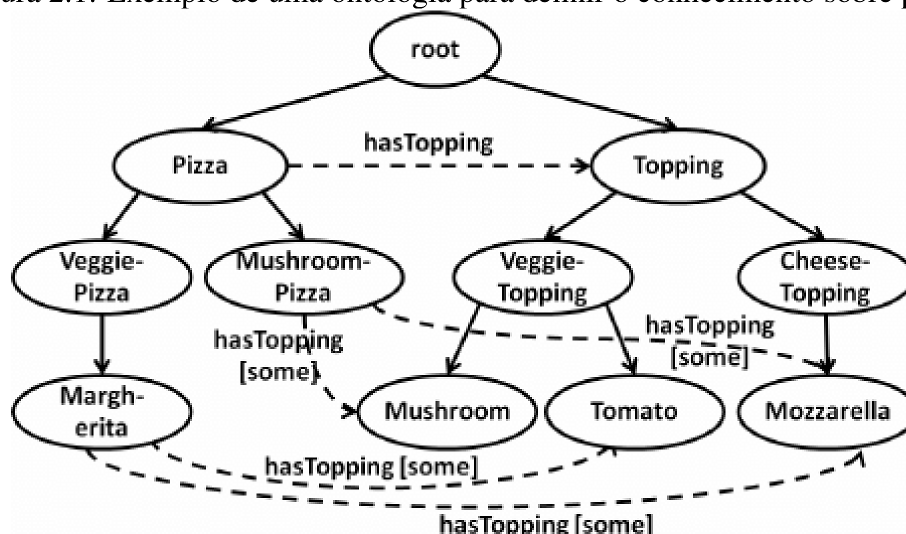
2.1 Ontologia

A teoria das ontologias encontra suas raízes na filosofia, onde uma ontologia é tradicionalmente definida como o estudo do ser, da existência e da natureza da realidade. O termo 'ontologia' foi cunhado, independentemente, em 1613 por dois diferentes filósofos: Rudolf Göckel e Jacob Lorhard (SMITH, 2012), e tem sua origem em outras duas palavras: *onto* que significa "o Ser" e *logia*, que significa "estudo ou conhecimento". Logo, o termo ontologia significa "o estudo ou conhecimento do Ser, dos entes ou das coisas tais como são em si mesmas, real e verdadeiramente" (SCHIESSL, 2007). Outras áreas do conhecimento, no entanto, se apropriaram do termo *Ontologia*, e seu sentido original da filosofia, e consequentemente criaram novos conceitos para o termo de acordo com a sua área de estudo. Uma das áreas de estudo que se apropriou do termo 'ontologia' e deu a ele uma dimensão mais prática é a ciência da computação. Para Gruber, uma ontologia é uma estrutura conceitual que busca representar, de maneira formal e organizada, o conhecimento sobre um domínio específico, destacando as relações entre os diversos elementos desse domínio (GRUBER, 1993b).

Aprofundando ainda mais o conceito de ontologia na ciência da computação, uma ontologia é uma representação formal e explicitamente definida de um domínio específico. Ela é composta por classes (ou conceitos), propriedades e relações entre essas classes, bem como indivíduos que são instâncias dessas classes (GUARINO; WELTY, 2009). A ontologia permite a captura e organização de conhecimento complexo em um formato compreensível por máquinas, estabelecendo uma base para a comunicação e compartilhamento de informações entre sistemas heterogêneos, a Figura 2.1 representa uma ontologia que apresenta a definição dos conceitos sobre pizza, como por exemplo: pizza, recheios, pizza de cogumelo, etc. Também apresenta relacionamentos entre diferentes conceitos, por exemplo: pizza margherita possui o recheio tomate e muçarela, e especializações de conceitos, por exemplo: recheio vegano é um tipo de recheio, e tomate é um tipo de recheio vegano. Esse tipo de representação, definição e formalização sobre áreas de domínio cada vez mais complexas é especialmente relevante na era digital, onde a quantidade de dados disponíveis cresce exponencialmente e é necessário encontrar maneiras eficientes de extrair significado e novos conhecimentos desses dados.

Dentre as aplicações na ciência da computação, ontologias desempenham um pa-

Figura 2.1: Exemplo de uma ontologia para definir o conhecimento sobre pizza



Fonte: Efficient regression testing of ontology-driven systems(KIM et al., 2012)

pel crucial em áreas como a inteligência artificial, a web semântica e a engenharia de conhecimento. Através da estruturação sistemática de conceitos e relações, as ontologias oferecem uma base sólida para a representação do conhecimento de maneira semântica, permitindo que máquinas entendam e raciocinem sobre o mundo de maneira mais próxima à capacidade humana. Elas também têm sido amplamente utilizadas em sistemas de recomendação, processamento de linguagem natural e na construção de sistemas especialistas (NOY; MCGUINNESS et al., 2001). O desenvolvimento e uso de ontologias tem sido enriquecido por uma variedade de linguagens e padrões, como OWL (*Web Ontology Language*) (MCGUINNESS; HARMELEN et al., 2004) e RDF (*Resource Description Framework*) (MCBRIDE, 2004), que proporcionam as ferramentas necessárias para modelagem, armazenamento e consulta de ontologias.

Na robótica, o uso de ontologias captura não apenas as características físicas do mundo ao redor, mas também os conceitos abstratos, relações espaciais e temporais, permitindo que os robôs tomem decisões informadas e executem tarefas de maneira mais eficiente.

Por exemplo, um robô autônomo que navega em um ambiente desconhecido pode utilizar uma ontologia para representar informações sobre obstáculos, caminhos livres, zonas de perigo e outros elementos relevantes para a navegação segura. Além disso, ontologias podem descrever interações entre objetos e agentes, permitindo que os robôs compreendam contextos sociais e colaborativos. Isso é particularmente valioso em situações onde robôs interagem com seres humanos ou outros robôs, como em ambientes de manufatura colaborativa ou assistência a idosos. Outro aspecto crucial é a capacidade dos

robôs de raciocinar sobre o conhecimento representado na ontologia. Isso implica que um robô pode inferir novas informações a partir do conhecimento existente na ontologia. Por exemplo, se um robô sabe que um objeto é inflamável e que há uma fonte de calor próxima, ele pode inferir que existe um risco aumentado de incêndio.

No entanto, a criação de ontologias não é uma tarefa trivial. Requer conhecimento profundo do domínio a ser modelado, habilidades em linguagens de ontologia como OWL e RDF, e uma compreensão das relações e hierarquias que regem o domínio. A evolução constante das ontologias também é um desafio, já que novos conhecimentos e mudanças no domínio exigem atualizações e revisões na estrutura da ontologia (NOY; MCGUINNESS et al., 2001).

2.2 Robótica e Ética

Robótica e ética são dois assuntos que, embora, tenham origens diferentes, muitas vezes, especialmente no imaginário coletivo, remetem um ao outro, e portanto são abordados e estudados quase que como extensão ao outro. Essa relação entre robótica e ética pode ser explicada em Frankenstein (SHELLEY; BOLTON, 2018) na obra uma criatura humanoide é construída por um cientista, e apesar de o monstro não apresentar nenhum indício de ser uma entidade maligna, as reações a ele são sempre carregadas de desconfiança, medo, e hostilidade até o ponto que essa criatura, farto dessas reações e mal tratado pela humanidade, se torna obsessivo na ideia de se vingar de seu criador. A essa reação que a humanidade tem ao monstro, foi cunhado o termo "Sindrome de Frankenstein", que é definido como "O medo de que uma criação feita por um ser humano se volte contra seu criador e destrua a humanidade"(ROLLIN, 1995).

Essa abordagem, da síndrome de Frankenstein, é capaz de explicar o porquê de o ser humano, instintivamente se preocupar e automaticamente analisar a robótica à luz da ética, o que pode ser observado em outros exemplos de obras da literatura clássica de ficção cientifica como "Eu, robô"(ASIMOV, 2004), onde diferentes contos abordam a relação dos próprios robôs com as três leis da robótica, propostas pelo próprio autor, e apresenta diversas situações que constituem verdadeiros dilemas éticos.

Na robótica, área de estudo da computação e engenharia, no entanto, essa preocupação surge não desse sentimento instintivo de que temos medo dos robôs, mas sim da necessidade de antever cenários e metodologias de trabalho a medida que temos robôs tomando decisões que afetam a vida humana e seu bem estar o tempo todo, assim como a

crescente utilização de robôs associados à técnicas de inteligência artificial, onde muitas vezes os mecanismos que explicam as tomadas de decisões não são transparentes nem ao desenvolvedor, nem ao humano que interage com a máquina. Dado isso, é importante que a área da robótica se antecipe às discussões que serão geradas pela sociedade quanto a vários aspectos éticos conforme essas tecnologias estejam cada vez mais presentes no cotidiano dessa sociedade. A seguir é apresentado uma breve revisão bibliográfica sobre outros estudos que aprofundam a discussão sobre robótica e ética, quanto a segurança, privacidade, implicações éticas, e mais.

Um dos temas mais discutidos é a segurança dos seres humanos em torno da interação com robôs. Em um estudo publicado por Culbertson and Gomila (2018), os autores discutem a necessidade de projetar robôs que sejam seguros para interagir com seres humanos e destacam a importância de considerar a segurança dos usuários ao projetar robôs autônomos. Os autores argumentam que os desenvolvedores de robôs têm uma responsabilidade ética de garantir que seus produtos sejam seguros e não causem danos a seres humanos ou ao meio ambiente.

Eles argumentam que a segurança é uma prioridade crítica na robótica, já que um erro pode levar a danos irreparáveis. Além disso, os desenvolvedores de robôs autônomos devem considerar a responsabilidade legal e moral pelos danos causados por seus produtos.

Culbertson e Gomila também discutem a importância da confiança em sistemas autônomos, argumentando que os usuários devem poder confiar na capacidade do robô de operar com segurança e confiabilidade. Eles também destacam a importância da privacidade na robótica, observando que a coleta e uso de dados pessoais por robôs podem violar a privacidade dos usuários e levar a problemas éticos.

De fato, a privacidade é outro tema que tem recebido atenção crescente. No artigo *Privacy and robots: Towards a framework for ethical design*, de Marina Powers, a autora explora as implicações da coleta de dados por robôs e discute a necessidade de políticas claras e regulamentação para proteger a privacidade dos usuários (POWERS, 2019).

Mais pontos importantes são as implicações éticas e sociais da automação de empregos. No artigo *The future of employment: How susceptible are jobs to computerisation?*, publicado por Osborne e Frey, os autores discutem como a automação afetará diferentes setores da economia e discutem as implicações éticas da redução do número de empregos disponíveis (OSBORNE; FREY, 2017).

Outra questão ética importante é a programação de robôs com inteligência ar-

tificial. Em um artigo de (ANDERSON; ANDERSON, 2019), os autores exploram a necessidade de programar valores éticos em robôs autônomos e discutem como isso pode ser alcançado.

Finalmente, a questão da utilização de robôs em conflitos militares também tem sido objeto de discussão. Em um artigo de (ARKIN, 2019), o autor discute as implicações éticas do uso de robôs em conflitos armados e destaca a importância de considerar as consequências de longo prazo do uso de robôs militares.

Esses são apenas alguns exemplos da vasta literatura que aborda as questões éticas da robótica. É importante que a pesquisa continue a explorar essas questões para garantir que os robôs sejam projetados e utilizados de maneira ética e responsável.

2.3 Primeiro padrão ontológico global para desenvolvimento de robôs e sistemas de automação que consideram aspectos éticos

Com o objetivo de antecipar a discussão quanto a construção de sistemas autônomos que consideram aspectos éticos, foi desenvolvido, entre 2017 e 2021, a primeira ontologia para o desenvolvimento de sistemas que consideram aspectos éticos, o padrão IEEE 7007 (PRESTES et al., 2021). Dentre os principais objetivos para o desenvolvimento do IEEE 7007, destacam-se:

- Antever contextos e casos de uso de futuras aplicações.
- Criar um relacionamento entre tecnologia e valores humanos.
- Endereçar o impacto da tecnologia, especialmente quando se trata de confiança entre humano e máquina, segurança da informação, proteção contra a vida, privacidade de dados e vieses algorítmicos.

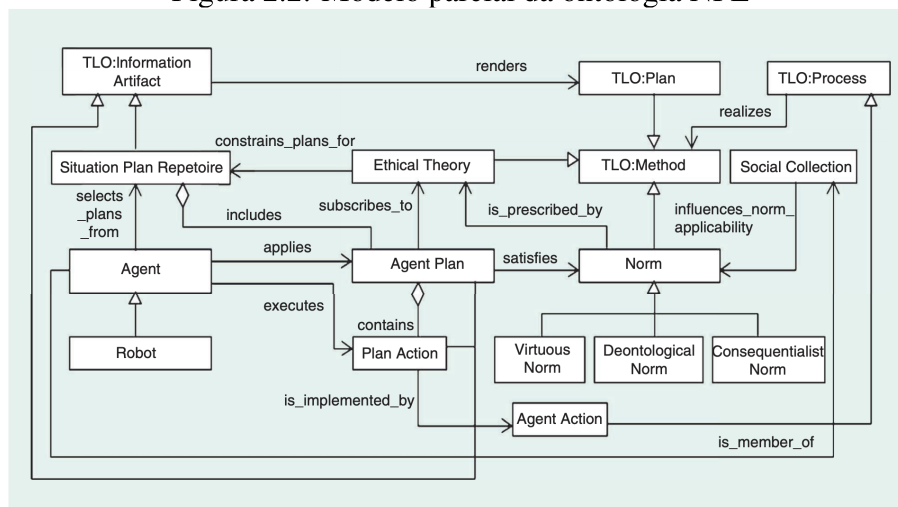
O padrão IEEE 7007 se constitui de um conjunto de ontologias, a ontologia de normas e princípios éticos (**NPE**), a de privacidade e proteção de dados (**PPD**), a de transparência e prestação de contas (**TPC**), e a de gestão de violações éticas (**GVE**).

2.3.1 Ontologia para normas e princípios éticos

O subdomínio da ontologia NPE formaliza as obrigações associadas às teorias e princípios éticos conforme ilustrado na Figura 2.2. Isso inclui axiomas para conceitos

como normas, teoria ética, repertorio de planos para situações, planos do agente e ações do agente, bem como seus relacionamentos.

Figura 2.2: Modelo parcial da ontologia NPE



Fonte: *The first global ontological standard for ethically driven robotics and automation systems* (PRESTES et al., 2021)

2.3.2 Ontologia para privacidade e proteção de dados

O subdomínio da ontologia **PPD** formaliza conceitos e relacionamentos entre agentes, entidades e organizações que estão envolvidos nos processos de coleta, processamento, transferência, armazenamento e retenção de dados onde sistemas autônomos estão inseridos. A ontologia PPD também inclui princípios como privacidade como design, proteção de dados como design, e direitos humanos como design.

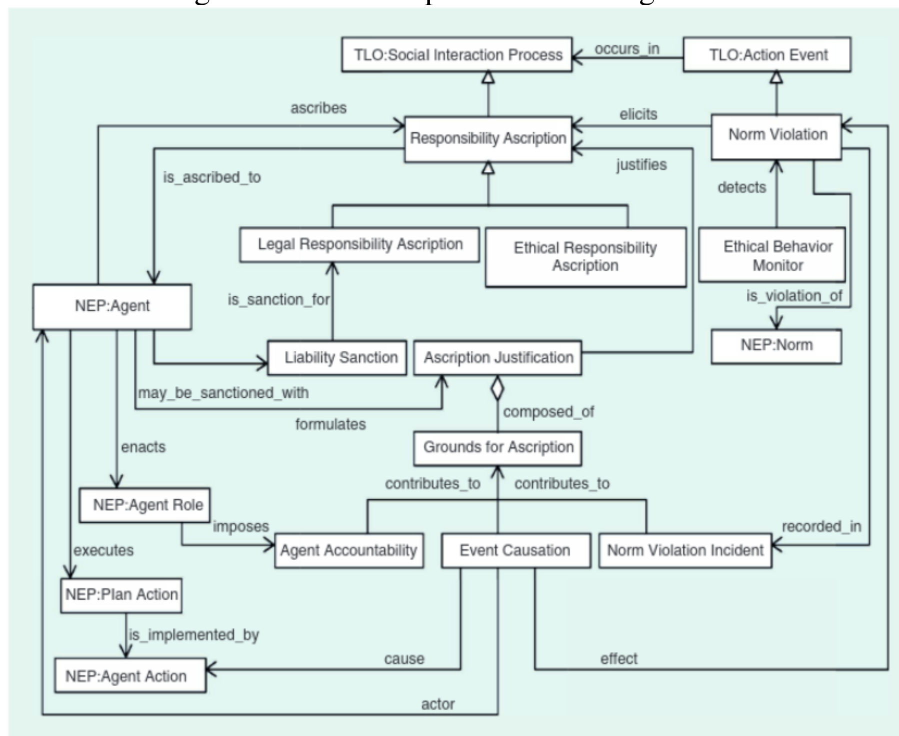
2.3.3 Ontologia para transparência e prestação de contas

O subdomínio da ontologia **TPC** formaliza o vocabulário e compromissos que habilitem sistemas autônomos éticos a terem a capacidade de prover explicações para planos e ações, de modo que os agentes sejam transparentes em suas interações com outros agentes.

2.3.4 Ontologia para gestão de violações éticas

O subdomínio da ontologia **GVE** formaliza a terminologia associada à capacidade de detectar, avaliar e gerenciar violações éticas e de normas acontecidas no agente ou geradas pelo comportamento autônomo do agente. Nesse subdomínio da ontologia que acontece a modelagem de conceitos como violação de norma, atribuição de responsabilidades, atribuição de responsabilidade de violações de norma, aplicação de penalidade por ser responsabilizado por uma violação, monitoramento de comportamento ético, dentre outros. Abaixo, na Figura 2.3 é apresentado uma representação gráfica parcial do subdomínio **GVE**.

Figura 2.3: Modelo parcial da ontologia GVE



Fonte: *The first global ontological standard for ethically driven robotics and automation systems* (PRESTES et al., 2021)

Por se tratar do subdomínio que será abordado com mais ênfase nesse trabalho, cabe aqui um detalhamento mais aprofundado desse subdomínio. A Tabela 2.1 apresenta um detalhamento de todas as classes, que estão definidas no subdomínio **GVE** seguido de uma breve descrição do que essa classe representa de acordo com as definições presentes na especificação IEEE 7007 (PRESTES et al., 2021).

Tabela 2.1: Classes da ontologia **GVE**

Classe	Descrição
EventCausation	Categoriza a causa de um evento, identificando um ator da classe `nep:Agent` e a ação desse agente associada a um evento de violação de norma.
ResponsibilityAscription	Subcategoria do processo de interação social (`tlo:SocialInteractionProcess`) que classifica entidades que atribuem responsabilidade por uma violação de norma para um agente.
AscriptionJustification	Subcategoria da classe artefato de informação (`tlo:InformationArtifact`) que classifica a coleta de fatos formulados e afirmados por um agente para atribuir responsabilidades por Violações de Normas éticas e legais.
GroundsForAscription	Subcategoria da classe artefato de informação (`tlo:InformationArtifact`) que classifica a coleta de circunstâncias factuais, eventos causais e obrigações legais ou éticas que são avaliadas para se tornarem a justificativa para atribuir responsabilidade por violações de normas.
AgentAccountability	Subcategoria da classe esquema (`tlo:Schema`) que classifica as propriedades do agente, como idade, estado físico e mental, habilidades, intenções, conhecimento, responsabilidades de papel e autoridade que contribuem para a avaliação da responsabilidade do agente ou agência por uma Violação de Norma.
NormViolationIncident	Subcategoria de artefato de informação (`tlo:InformationArtifact`) que classifica entidades que documentam e registram ocorrências de violação de normas.
EthicalBehaviorMonitor	Subcategoria da classe objeto (`tlo:Object`) que classifica um agente ou agente componente do sistema que monitora sistemas de **IA** quanto à conformidade com o comportamento ético normativo.
NormViolation	Subcategoria da classe evento de ação (`tlo:ActionEvent`) que classifica entidades que indicam a falha de um agente em conformar-se às regras de comportamento de uma norma relevantes para a situação do agente.
SocioTechnologyGovernance	Subcategoria da classe processo de interação (`tlo:InteractionProcess`) que classifica os esforços de agências governamentais para fornecer supervisão e gestão dos processos sociais e tecnológicos que criam, modificam e sustentam o design e a introdução de artefatos e métodos envolvidos em sistemas que abrangem aspectos tanto tecnológicos quanto sociológicos.
LiabilitySanction	Subcategoria da classe método (`tlo:Method`) que classifica entidades designando punições ou penalidades relevantes definidas por uma agência autoritária que podem ser impostas contra um agente ao qual é atribuída responsabilidade por uma violação de norma legal.
LegalResponsibilityAscription	Subcategoria da classe `evm:ResponsibilityAscription` que classifica entidades de processo que atribuem responsabilidade por uma violação de norma legal.
EthicalResponsibilityAscription	Subcategoria da classe `evm:ResponsibilityAscription` que classifica entidades de processo que atribuem responsabilidade por uma violação de norma ética.

2.3.5 Possíveis casos de uso do padrão ontológico

Dentre os exemplos de cenários em que o padrão ontológico pode ser utilizado, o padrão IEEE 7007 (PRESTES et al., 2021) apresenta algumas situações onde diferentes aplicações robóticas têm seu comportamento construído com o apoio do padrão ontológico. São cenários como: (a) um robô assistente doméstico, que cuida de um humano com princípios de demência, e percebe um padrão de movimentações financeiras de alto valor saindo da conta do humano e indo para a conta de um familiar, ao questionar o humano ele recebe uma informação, e ao questionar o familiar, recebe uma explicação diferente. Diante de duas informações diferentes, e de diagnosticar uma situação além da sua capacidade, ele sugere ao humano que chamem uma assistente social para lidar com a situação; e (b) um robô assistente pessoal, que auxilia um humano, que tem no seu com-

portamento a privacidade e proteção dos dados do usuário como uma obrigação, e que tem a capacidade e a obrigação de explicar e justificar suas ações quando questionado. Um dia, após o usuário ter tomado os seus remédios, que foram entregues pelo robô, a pessoa passa mal e o robô decide chamar uma equipe médica. Quando os médicos chegam, a pessoa está inconsciente e então perguntam ao robô quais medicamentos a pessoa havia tomado, e visando proteger a privacidade do usuário, o robô se recusa a compartilhar a informação. Posteriormente, porém, ao ser questionado sobre o porquê de não ter colaborado com os médicos, o robô é capaz de detalhar todos os motivos de ter agido dessa forma. Possibilitando até que os projetistas, a partir desse diagnóstico, possam rever o comportamento do robô.

2.4 Programação orientada a ontologias

A Programação Orientada a Ontologias (POO) é um campo interdisciplinar que combina os princípios da programação orientada a objetos com o uso de ontologias. Ontologias são modelos conceituais que descrevem as relações entre os diferentes objetos e conceitos de um domínio específico. A POO busca utilizar essas ontologias para aprimorar o desenvolvimento de software, facilitando a representação, organização e manipulação de informações (MUSEN et al., 2000). A POO é aplicada em diversas áreas, como engenharia de software, sistemas de informação, web semântica, inteligência artificial, robótica e recuperação de informações. A seguir, é apresentado uma breve revisão bibliográfica de estudos relevantes sobre programação orientada a ontologias.

O livro *Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web* (GÓMEZ-PÉREZ; FERNÁNDEZ-LÓPEZ; CORCHO, 2006) fornece exemplos práticos de como aplicar engenharia ontológica em diferentes domínios, como gerenciamento de conhecimento, comércio eletrônico e Web Semântica. Ele discute como a engenharia ontológica pode ser utilizada para melhorar a gestão do conhecimento em organizações, facilitando a captura, organização e recuperação de informações relevantes. Além disso, explora como as ontologias podem ser aplicadas no comércio eletrônico para aprimorar a busca por produtos, personalizar recomendações e facilitar a interação entre sistemas de diferentes fornecedores.

Uma parte significativa do livro é dedicada à aplicação das ontologias na Web Semântica. A Web Semântica é uma extensão da Web tradicional que visa adicionar significado aos dados, permitindo que máquinas compreendam e processem informações

de maneira mais inteligente (BERNERS-LEE; HENDLER; LASSILA, 2001). As ontologias desempenham um papel crucial na construção dessa Web Semântica, fornecendo uma estrutura para representar conceitos e relacionamentos entre eles.

O livro *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL* (ALLEMANG; HENDLER, 2011) é uma referência para entender as bases da Web Semântica e como aplicar ontologias em sistemas de informação. Ele explora como usar as linguagens de descrição de ontologias, RDFS e OWL, para representar e manipular dados de forma semântica.

O livro *Ontologies: A Handbook of Principles, Concepts and Applications in Information Systems* (KISHORE; RAMESH, 2007) abrange uma ampla gama de tópicos relacionados a ontologias, e como construir sistemas computacionais através de técnicas que incluem modelagem de ontologias, linguagens de ontologia, construção de ontologias, além de explorar suas aplicações em sistemas de informação e outras áreas.

Em *Ontology-Based Applications for Enterprise Systems and Knowledge Management* (AHMAD, 2012), o estudo do livro se concentra nas aplicações práticas de ontologias em sistemas empresariais e gerenciamento de conhecimento. Ele explora como as ontologias podem ser usadas para melhorar a organização e recuperação de informações em ambientes corporativos.

No livro *Programming with the Semantic Web* (HEBELER et al., 2011), o autor se aprofunda nas técnicas de programação orientada a ontologias, com foco na utilização de linguagens de consulta como SPARQL e no desenvolvimento de aplicativos semânticos.

2.4.1 OWL

OWL, sigla para Web Ontology Language, é uma linguagem de modelagem semântica projetada para representar conhecimento rico e complexo sobre um domínio específico. É uma das principais linguagens da Web Semântica, que tem como objetivo tornar os dados na web mais legíveis tanto para humanos quanto para máquinas (BERNERS-LEE; HENDLER; LASSILA, 2001). Através da representação formal de ontologias, o OWL possibilita que computadores entendam e raciocinem sobre os relacionamentos entre os diferentes conceitos em um domínio.

A documentação oficial do W3C (World Wide Web Consortium) (OWL. . . , 2012) para OWL é uma fonte autoritária e abrangente para entender os diferentes recursos da linguagem, suas construções e semântica. A especificação é frequentemente atualizada e

mantida pelo grupo de trabalho de Web Semântica do W3C.

2.4.2 Owlready2

Owlready2 é uma biblioteca para a linguagem de programação Python, criada para trabalhar com ontologias OWL (Web Ontology Language) de forma eficiente, possibilitando a programação orientada a ontologia (POO). Ela oferece uma série de funcionalidades para criar, manipular e consultar ontologias OWL em um ambiente de programação Python. A biblioteca foi projetada para simplificar tarefas relacionadas à ontologia, permitindo que os desenvolvedores integrem conceitos semânticos em suas aplicações de forma mais fácil (LAMY, 2017).

No artigo que apresenta a Owlready2 (LAMY, 2017), o autor e criador da ferramenta explora a funcionalidade da biblioteca e como ela pode ser utilizada para criar ontologias, carregar ontologias existentes, criar instâncias de classes, realizar consultas SPARQL e muito mais. O artigo se concentra especialmente em como o Owlready2 é aplicável em ontologias biomédicas, o que destaca sua versatilidade em diversos domínios.

Já no livro *Ontologies in Python with Owlready2* (JEAN-BAPTISTE, 2021), o autor e criador do Owlready2 apresenta um tutorial prático que introduz os conceitos básicos de ontologias, OWL e como usar o Owlready2 para manipular ontologias em Python. Ele guia os leitores passo a passo na criação de uma ontologia simples, na definição de classes e propriedades, na criação de instâncias e na realização de consultas a indivíduos, conceitos e classes pertencentes à ontologia.

3 METODOLOGIA

Este estudo tem como proposta a criação de uma aplicação focada na análise de eventos robóticos a partir de uma perspectiva ontológica. Dentre os padrões ontológicos definidos no trabalho, a aplicação criada focará sobretudo na ontologia de gestão de violações éticas (GVE). A escolha de se fazer a análise de eventos utilizando somente essa ontologia, e portanto deixando de lado as ontologias de normas e princípios éticos, a de privacidade e proteção de dados e a de transparência e prestação de contas se dá, especialmente, devido à complexidade existente nos processos de criar uma aplicação que utilize o padrão ontológico completamente. Uma das maiores dificuldades envolvidas na criação de uma aplicação tão abrangente é a criação de cenários que compreendam todas as entidades e relacionamentos presentes nesses padrões ontológicos e que não seja apenas um cenário que contenha esses elementos apenas por conter, e sim modele um cenário significativo, onde a análise dos eventos sirva para reproduzir cenários do mundo real. Outro motivo de se utilizar apenas a ontologia GVE na análise é que com o escopo reduzido, fica mais fácil avaliar limitações, dificuldades, e também os potenciais benefícios, para que em um próximo momento esse estudo sirva de base para que sistemas que utilizem toda a ontologia sejam criados.

3.1 Sistema desenvolvido

O sistema proposto por esse trabalho possui as seguintes funcionalidades:

- Carregar a estrutura da ontologia GVE a partir de arquivos OWL (MCGUINNESS; HARMELEN et al., 2004) que a descrevem.
- Carregar indivíduos da ontologia a partir de um arquivo CSV (SHAFRANOVICH, 2005).
- Carregar relacionamentos entre indivíduos da ontologia a partir de um outro arquivo CSV.
- Sobre esses indivíduos e relacionamentos carregados inicialmente, realizar a inferência lógica de informações que não estavam nos dados iniciais.
- Disponibilizar ao usuário um CLI (*Command line interface*) para navegar sobre os dados iniciais e inferidos, permitindo operações de busca, visualização, etc.

3.1.1 Tecnologias Utilizadas

O sistema foi totalmente criado utilizando a linguagem de programação Python, versão 3.8. Essa escolha se deu não tanto por características específicas da linguagem, mas sim pela existência da biblioteca para manipulação de ontologias Owlready2. A escolha de se utilizar a linguagem, no entanto, trouxe vários aspectos positivos para a solução final, como a facilidade para realizar o carregamento de arquivos CSV (*comma separated values*), de criação de um aplicativo CLI (*Command line interface*), e de modelar a solução utilizando uma abordagem orientada a objetos. Esses aspectos e funcionalidades são nativos à linguagem, e não demandaram instalação de bibliotecas adicionais.

3.1.2 Aplicação da ferramenta desenvolvida

O sistema criado nesse trabalho tem como objetivo a visualização de entidades pertencentes à ontologia **GVE**, e para possibilitar isso, ele deve funcionar como uma ferramenta que possibilite, para alguém familiarizado com a ontologia **GVE**, modelar um cenário em cima da mesma. Nesse contexto, um cenário refere-se a um conjunto de instâncias (ou indivíduos) das classes definidas pela ontologia **GVE**, e seus relacionamentos.

Ao mesmo passo que o aplicativo deve possibilitar a instanciação de qualquer conjunto de indivíduos pertencentes à ontologia, ele deve oferecer ao usuário um conjunto de ferramentas que o permita realizar algumas operações sobre essas instâncias e relacionamentos modelados, especialmente possibilitando ao usuário visualizar as propriedades de cada um dos indivíduos instanciado, navegar entre os vários relacionamentos estabelecidos entre as entidades, buscar por relações entre diferentes classes e indivíduos e visualizar propriedades que o próprio padrão ontológico gerou (ou "raciocinou", segundo termo utilizado dentro da área de ontologias) sobre os dados iniciais.

3.1.2.1 Criação de cenários que explorem a utilização dos conceitos éticos

Um padrão ontológico define **classes**, que são as entidades ou conceitos dentro de um domínio, **propriedades**, que são os relacionamentos entre essas diferentes classes, **indivíduos**, que são as diferentes instâncias das classes, e **axiomas** (ou regras), que determinam as regras ou restrições dentro de um domínio (GRUBER, 1993a). Quando um usuário for utilizar a aplicação construída nesse trabalho, uma parte destas definições já

estarão prontas, como as definições de classe e dos axiomas. Essas duas definições vem do arquivo OWL que define a ontologia de **GVE**.

O usuário do aplicativo, portanto, deve providenciar à aplicação quais os indivíduos e relacionamentos que existirão nessa execução do programa. Essa coleção de indivíduos e relacionamentos disponibilizados inicialmente pelo usuário ao aplicativo é denominado neste trabalho de **cenário**.

3.1.2.2 Visualização dos resultados

Após o carregamento de um cenário para o programa, é importante que o aplicativo disponibilize maneiras para que os dados imputados sejam visualizados. Para isso, o programa disponibilizará mecanismos para visualizar todos os indivíduos carregados pelo usuário, indivíduos que foram raciocinados por axiomas da ontologia, as propriedades de cada indivíduo, incluindo seus relacionamentos com outras entidades da ontologia, e relacionamentos que foram raciocinados automaticamente de acordo com os axiomas presentes na ontologia. Por fim, o sistema também disponibilizará um sistema de navegação e busca dentro de todos os dados do sistema. Por exemplo, o usuário poderá buscar por todos os incidentes de violação ética (uma das classes presentes na ontologia **GVE**) que foram causados por um robô (outra entidade presente no padrão ontológico), e visualizar quais são os relacionamentos que ligam essas duas entidades.

4 IMPLEMENTAÇÃO

Nesse capítulo será apresentado em maiores detalhes como o sistema proposto no capítulo anterior foi construído, como funciona a entrada de dados, o sistema de visualização de resultados, além de disponibilizar um guia de utilização do programa. Antes de entrar nos detalhes de implementação, é importante salientar que todas as técnicas descritas aqui, assim como as funcionalidades desenvolvidas são de certa maneira pioneiras, e com exceção do módulo de carregamento de ontologias **OWL** e de raciocínio que são fortemente apoiadas na utilização da biblioteca *Owlready2*, todo o restante foi totalmente desenvolvido utilizando recursos nativos da linguagem, sem a utilização de bibliotecas de terceiros.

O programa trata-se de um aplicativo de linha de comando, desenvolvido em python, que chamamos de `evm_loader.py`. Ao executar o programa estamos carregando a ontologia **GVE** na memória. Na chamada do programa também é preciso indicar uma lista de arquivos de indivíduos do cenário e uma lista de arquivos de relacionamentos entre indivíduos deste cenário.

Para maiores detalhes, ver Apêndice A, que traz um manual detalhado da ferramenta proposta.

Ao se executar o programa passando os arquivos que contém a lista de indivíduos e propriedades da ontologia, o resultado da execução é que o aplicativo carrega a ontologia **GVE** a partir de arquivos .OWL, carrega os indivíduos e propriedades passadas pelo usuário e logo em seguida executa o raciocínio dos dados, (ou *reasoning*), gerando novos conhecimentos sobre os dados, podendo reclassificar indivíduos, ou gerar novos relacionamentos a partir dos axiomas definidos no padrão ontológico. Após essa etapa de raciocínio, o programa exibe na tela uma lista com todos os indivíduos e suas propriedades, que podem ser de dois tipos:

- propriedade direta
  Uma propriedade direta de um indivíduo é aquela cujo domínio da propriedade é a classe do indivíduo. Por exemplo, na ontologia **GVE** a classe `evmNormViolation` possui a propriedade `evmis_violation_of`, que deve apontar para uma instância da classe `nepNorm`. Dizemos então que `evmis_violation_of` é uma propriedade direta de um indivíduo da classe `evmNormViolation`.
- propriedade inversa
  Uma propriedade inversa de um indivíduo é aquela cujo domínio da propriedade

é outra classe, porém o contradomínio, ou imagem, do relacionamento é a classe desse indivíduo. Por exemplo, na ontologia **GVE** a classe `evmEventCausation` possui a propriedade `evmeffect`, que deve apontar para uma instância da classe `evmNormViolation`. Dizemos então que `evmeffect` é uma propriedade inversa de um indivíduo da classe `evmNormViolation`.

Com isso, temos que as execuções anteriores do programa `evm_loader.py` gerariam um relatório na linha de comando listando todos os indivíduos desse cenário, listando para cada indivíduo suas propriedades diretas e suas propriedades inversas, e logo após o programa encerraria. Mais informações sobre essa visualização da ontologia estão disponíveis na seção 4.8.

Caso o usuário queira realizar uma navegação interativa sobre esses indivíduos, visualizando relacionamentos e navegando entre indivíduos relacionados, é possível habilitar a opção `-cli`. Nela, o programa realiza o carregamento da ontologia OWL e o carregamento dos indivíduos e propriedades indicados pelo usuário normalmente. Porém, ao invés de exibir as informações de cada um dos indivíduos após o carregamento, é iniciado o aplicativo de navegação por linha de comando, que possibilita a navegação iterativa entre indivíduos da ontologia, a busca por relacionamentos e a exibição de informações ontologia. Mais informações sobre o modo de linha de comando estão disponíveis na seção 4.9.

Outra opção que o usuário pode passar na chamada do programa é o parâmetro `-v`, ou `-verbose`, com isso o programa executará no modo verboso, e exibirá para o usuário informações sobre o carregamento da ontologia e sobre inferências realizadas. Além disso, o modo verboso possui 2 níveis, no primeiro apenas informações de inferência e erros de validação da ontologia são exibidos, no nível dois toda a ontologia carregada é exibida na tela.

4.1 Carregamento da ontologia OWL

Independente das opções dadas pelo usuário do aplicativo, uma funcionalidade que sempre é realizada pelo programa é a de carregar a ontologia **GVE** para a memória, para que aí então os dados e funcionalidades requisitadas pelo usuário possam ser definidas e executadas. Esse carregamento é feito utilizando a biblioteca Owlready2 (ver 2.4.2), uma biblioteca que permite a programação orientada a ontologias utilizando a linguagem

de programação Python.

Para que o carregamento da ontologia **GVE** possa ser realizada pela biblioteca, é necessário a existência de todos os três seguintes arquivos **OWL**: `evm.owl`, `nep.owl` e `tlo.owl`. A necessidade dos arquivos `nep.owl` e `tlo.owl` para carregar a ontologia **GVE** se dá pois as classes definidas nessa ontologia possuem relacionamentos com classes das ontologias **NPE** e **OAN**, e portanto o carregamento do arquivo `evm.owl` individualmente não é suficiente para carregar toda a abrangência das definições do padrão ontológico. Na Figura 2.3, na seção 2.3.4, é exibido a modelagem parcial do subdomínio **GVE** da ontologia, nessa imagem é possível enxergar classes pertencentes à **GVE** que possuem relacionamentos com classes definidas nos subdomínios **NPE** e **OAN**.

Esses arquivos são resultados do trabalho *The first global ontological standard for ethically driven robotics and automation systems* (PRESTES et al., 2021), e não sofreram nenhuma alteração para serem utilizados nesse trabalho.

4.2 Acessando as classes da ontologia

As classes da ontologia carregada com a biblioteca Owlready2 estão disponíveis através do elemento `ontology.world.classes()` cujo conteúdo é uma lista que contém o nome de todas as classes definidas na ontologia:

```
1 [.evmEventCausation, .evmResponsibilityAscription,
2 .evmAscriptionJustification, .evmGroundsForAscription,
3 .evmAgentAccountability, .evmNormViolationIncident,
4 .evmEthicalBehaviorMonitor,  ...]
```

4.3 Acessando os indivíduos da ontologia

Os indivíduos da ontologia carregada com a biblioteca Owlready2 estão disponíveis através do elemento `ontology.world.individuals()`, que contém uma lista com todos os indivíduos existentes.

Como os arquivos que são carregados nesse trabalho são os que definem o padrão ontológico **GVE**, e estão sendo utilizados diretamente sem nenhuma edição, é de se esperar que não existisse nenhum indivíduo já instanciado na ontologia. No entanto, alguns indivíduos estão sempre presentes já nessa definição, e por isso quando lemos a lista de

indivíduos após realizar o carregamento dos arquivos OWL, temos o seguinte resultado:

```
[.evmmulti_agents, .evmcertified_high_capacity,
.evmevolving_capacity, .evmno_capacity,
.evmsingle_agent, .nephuman_directed, .nepethical_norm,
.neplegal_norm, .nepactivated, .nepexpired, .nepfulfilled,
.nepnot_applicable, .nepself_directed_team,
.nepsuspended, .nepviolated]
```

Esses indivíduos preexistentes na ontologia são valores de enumerações definidas na ontologia **GVE** e podem acabar poluindo a visualização quando o usuário do aplicativo quiser instanciar seus próprios indivíduos. Por esse motivo, portanto, foi decidido por se deletar todos os valores que não forem utilizados no cenário definido pelo usuário.

4.4 Instanciando classes e criando indivíduos

Para se instanciar novos indivíduos utilizando as definições de classes da ontologia carregada com a biblioteca Owlready2, é necessário primeiro construir uma definição em Python dessa classe, utilizando sempre o mesmo nome utilizado na ontologia. Por exemplo, para se instanciar um indivíduo da classe `nepRobot`, é preciso definir uma classe Python `class nepRobot`, que herde as propriedades da classe definida pela biblioteca Owlready2 `owlready2.Thing`. Dentro do ambiente do Owlready2, todas as classes são subclasses dessa classe `owlready.Thing` (JEAN-BAPTISTE, 2021).

Para o aplicativo, foi criado um módulo chamado `Factory`, que reúne em uma única classe Python os mecanismos para a instanciação de todas as classes da ontologia **GVE**. Com a utilização desse módulo `Factory`, não é necessário criar de forma redundante todas as definições de classe, pois esse módulo abstrai isso.

Abaixo, um exemplo ilustra a utilização do módulo `Factory`, criado para essa aplicação, para realizar a instanciação de indivíduos.

```
import owlready2
from factory import Factory

ontology = owlready2.get_ontology("evm.owl").load()
factory = Factory(ontology)

```

```
factory.nepRobot('robot1')
factory.evmNormViolation('violation1')
```

Além da utilização apresentada acima, esse objeto factory também disponibiliza um método para que se adquira a função instanciadora de uma classe a partir do nome da mesma, o que se torna particularmente útil quando o usuário for criar indivíduos a partir de um arquivo de entrada. Abaixo é ilustrada essa utilização do objeto `factory`:

```
import owlready2
from factory import Factory

ontology = owlready2.get_ontology("evm.owl").load()
factory = Factory(ontology)

individual_creation_function = factory.get_factory_by_name('nepRobot')
individual_creation_function('robot1')
individual_creation_function =
        factory.get_factory_by_name('evmNormViolation')
individual_creation_function('violation1')
```

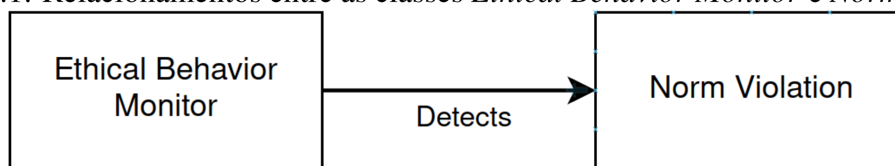
4.5 Definindo relacionamentos entre indivíduos

Para se definir relacionamentos (ou propriedades) entre os indivíduos pertencentes à ontologia carregada pela biblioteca Owlready2, é necessário ter antes os indivíduos envolvidos propriamente instanciados, para que aí então seja construido o relacionamento entre os dois. Por exemplo, a propriedade *Detects*, entre as classes ***Ethical Behavior Monitor*** e ***Norm Violation***, ilustrada na Figura 4.1, deve ser criada a partir do indivíduo cuja classe pertence ao domínio da propriedade, e apontada para o indivíduo cuja classe está no contradomínio dessa propriedade. Para o exemplo da propriedade *Detects*, o trecho de código que define essa propriedade para dois indivíduos das classes ***Ethical Behavior Monitor*** e ***Norm Violation*** é apresentado abaixo.

```
ethical_behavior_monitor=evmEthicalBehaviorMonitor("ebm_1")
norm_violation=evmNormViolation("nv_1")

ethical_behavior_monitor.evmdetects.append(norm_violation)
```

Figura 4.1: Relacionamentos entre as classes *Ethical Behavior Monitor* e *Norm Violation*



Fonte: O autor

Outro aspecto importante de se notar é a a utilização do método `.append()` para criar esse relacionamento. A utilização desse método denota que a propriedade é armazenada na biblioteca Owlready2 como uma lista, o que possibilita que o indivíduo `individuo_1` possua múltiplos relacionamentos do tipo `prop` com diferentes indivíduos. Como no exemplo abaixo, utilizando as classes da ontologia **GVE**, onde uma instância de um monitor de comportamento ético detecta diferentes violações de norma:

```
1 ethical_behavior_monitor.evmdetects.append(norm_violation1)
2 ethical_behavior_monitor.evmdetects.append(norm_violation2)
3 ethical_behavior_monitor.evmdetects.append(norm_violation3)
```

4.6 Raciocinar sobre os dados da ontologia

A partir do momento em que a ontologia está carregada pela biblioteca Owlready2, é possível realizar o processo de raciocínio (ou *reasoning*). Esse processo envolve a capacidade de derivar implicitamente novas informações a partir das relações e restrições definidas explicitamente na ontologia. Ou seja, mesmo que algumas informações não estejam diretamente declaradas, o raciocínio permite inferir essas informações com base nas regras e axiomas estabelecidos na ontologia. Isso inclui a inferência de hierarquias de classes, restrições sobre propriedades, classificação de instâncias e detecção de inconsistências na ontologia.

A biblioteca Owlready2 utiliza como *reasoner* a ferramenta *HermiT* (GLIMM et al., 2014). O *HermiT* é um *reasoner* (motor de raciocínio) de ontologias OWL desenvolvido pela Universidade de Oxford. E seu funcionamento é totalmente integrado com a biblioteca Owlready2, e pode ser executado conforme o exemplo abaixo:

```
1 import owlready2
2 ontology = owlready2.get_ontology("evm.owl").load()
```

```
3
4 # ... Cria individuos
5
6 # Realiza o raciocinio.
7 owlready2.sync_reasoner_hermit()
```

4.7 Carregamento de indivíduos e propriedades

Anteriormente foi explicado, de forma geral, como utilizar a biblioteca Owlready2 para carregar, manipular e visualizar ontologias, e mais especificamente, nas seções 4.4 e 4.5 foram abordados, respectivamente, mecanismos para a criação de indivíduos e relacionamentos. Esta seção, entretanto, busca apresentar a solução utilizada no aplicativo, que abstrai os mecanismos da biblioteca Owlready2 e possibilita que o usuário do aplicativo instancie múltiplos indivíduos e relacionamentos de forma simplificada.

Quando um usuário utiliza o aplicativo, conforme detalhado na seção A.1, o usuário está sempre carregando a ontologia **EVM** e passando ao programa um conjunto de entrada que se constituí de indivíduos e propriedades. Nesse trabalho, será denominado "cenário" um conjuntos de indivíduos e propriedades que é passado como entrada para o programa.

Para que o usuário possa criar um cenário para o programa, ele primeiro deve modelar esses cenários dentro de dois arquivos CSV(*Comma Separated Value*), o primeiro arquivo explicitando quais são os indivíduos do cenário modelado, e um segundo arquivo explicitando quais são os relacionamentos entre esses indivíduos.

Abaixo, na Tabela 4.1 é apresentado o formato de um arquivo que pode ser carregado pelo aplicativo para criar indivíduos em cima da ontologia **GVE**.

Tabela 4.1: Indivíduos de uma ontologia de gestão de violações éticas

Classe	Nome indivíduo
nepRobot	robot A
nepNorm	Not allowed to touch the walls
evmNormViolation	hit the wall
evmEthicalBehaviorMonitor	monitor 1
evmResponsibilityAscription	robot A responsibility ascription

Fonte: O autor

Sobre a Tabela 4.1, alguns pontos importantes para se destacar são que: a primeira coluna define qual é a classe do indivíduo, e a segunda coluna define o nome desse

indivíduo, sendo que esse nome servirá como um identificador, e portanto deve ser único.

O nome da classe é composta por duas partes: um prefixo que indica em qual padrão ontológico ela está definida, podendo ser **EVM**, que indica que vem do padrão *Ethical Violation Management*, ou padrão ético de gestão de violações éticas (GVE), **NEP**, que indica que é uma classe definida no padrão ontológico *Norms and Ethical Principles*, ou padrão de normas e princípios éticos (NPE), ou ainda **TLO**, indicando que é uma classe definida na ontologia *Top Level Ontology*, ou ontologia de alto nível (OAN) e ainda é *case sensitive*, e portanto o arquivo que contém os indivíduos deve conter o nome da classe exatamente como escrito acima. Uma listagem completa de todas as classes possíveis de serem instanciadas está disponível no Apêndice B.

Além do arquivo de indivíduos, o aplicativo também aceita um segundo arquivo, que contém a definição dos relacionamentos e as propriedades dos indivíduos criados anteriormente. Abaixo, na Tabela 4.2, temos um exemplo do formato de um arquivo CSV que define propriedades entre os indivíduos criados pelo arquivo da Tabela 4.1.

Tabela 4.2: Relacionamento entre indivíduos de uma ontologia de gestão de violações éticas

Propriedade	Indivíduo base	Indivíduo alvo
evmis_violation_of	hit the wall	Not allowed to touch the walls
evmelicits	hit the wall	robot A responsibility ascription
evmis_ascribed_to	robot A	robot A responsibility ascription
evmascribes	robot A responsibility ascription	robot A

Fonte: O autor

Sobre a estrutura da Tabela 4.2, cada linha dessa tabela é a representação de um relacionamento entre dois indivíduos, e dentro dessa linha as três colunas representam, respectivamente, o nome da propriedade, o nome do indivíduo da classe que pertence ao domínio da propriedade, e o nome do indivíduo da classe que pertence ao contradomínio da propriedade. O nome dos indivíduos devem ser os mesmos criados na segunda coluna da Tabela 4.2. Novamente, está disponível no Apêndice B uma listagem com todas as classes disponíveis para serem criadas, assim como todas as propriedades.

Internamente, o carregamento desses dois arquivos é realizado por dois módulos separados, o primeiro é responsável por realizar o carregamento do arquivo de indivíduos, e se chama `IndividualsLoader`, e um segundo módulo, responsável por realizar o carregamento do arquivo de propriedades, chamado `PropertiesLoader`.

O módulo `IndividualsLoader` trabalha apoiado em cima do módulo `Factory`, apresentado na seção 4.4. Para instanciar esse novo indivíduo, é realizado o seguinte procedimento para cada linha do arquivo de indivíduos.

```
class_name = csv_line[0]
individual_name = csv_line[1]

instantiator = factory.get_factory_by_name(class_name)
instantiator(individual_name)
```

Já o módulo `PropertiesLoader` executa o processo de localizar o indivíduo da classe domínio, que já foi criado, e localizar o indivíduo da classe contradomínio, também já criado, e os relacionar através do relacionamento. Para isso ser feito, é necessário lembrar que cada linha do arquivo de propriedades contém, respectivamente, os três itens: o nome da propriedade, o nome do indivíduo da classe do domínio, e o nome do indivíduo da classe do contradomínio. Para cada linha desse arquivo, é executado o procedimento de buscar os indivíduos já instanciados pelo nome, então localizar a propriedade do individuo pertencente a classe de domínio, e em seguida realizar o `.append()`, conforme apresentado na seção 4.5, do indivíduo da classe de contradomínio. Esse procedimento é feito para cada linha do arquivo de propriedades, e é representado pelo código abaixo:

```
property_name = csv_line[0]
source_individual = get_ontology_individual_by_name(csv_line[1])
target_individual = get_ontology_individual_by_name(csv_line[2]

ind_property = getattr(source_individual, property_name)
ind_property.append(target_individual)
```

4.8 Visualização dos indivíduos e propriedades

Quando o aplicativo é chamado, o usuário passa por argumento de linha de comando quais são os indivíduos e propriedades que compõe um cenário. Se nenhum outro argumento além dos arquivos de indivíduos e propriedades for informado, o comportamento padrão do programa é de listar e exibir informações sobre todos os indivíduos e suas propriedades.

A visualização de um indivíduo segue o seguinte formato:

```
========================================

```

```
Name: {nome do indivíduo}
Instance of: {nome da classe do indivíduo}

Properties:
{lista todas as propriedades diretas desse indivíduo, no formato
nome_desse_individuo -> propriedade -> nome do indivíduo alvo}

Inverse properties:
{lista todas as propriedades inversas do indivíduo
(aquelas que tem esse indivíduo como alvo, no formato
nome_de_um_outro_individuo -> propriedade -> nome_desse_individuo}


========================================
```

Quando essa visualização é realizada pelo aplicativo, o motor de raciocínio *HermiT* já foi executado, portanto é possível verificar através dessa visualização a existência de propriedades criadas pelo *HermiT*, ou até mesmo a inferência de classes. Vejamos o seguinte exemplo de uma execução completa, ilustrando os arquivos CSV criados e a execução do programa.

Arquivo **individuos.csv**:

```
nepRobot,robot
evmAscriptionJustification,"ascription justification"
```

Arquivo **propriedades.csv**:

```
tloformulates,"ascription justification","robot"
```

Carregamento e visualização desse cenário:

```
> ./evm_loader.py -i individuos.csv -p propriedades.csv
...
========================================

Name: robot
Instance of: nepRobot

```

```
8  Properties:
9  evm.robot -> tlo.tloformulatedBy -> evm.ascription justification
10
11 Inverse properties:
12 evm.ascription justification -> tlo.tloformulates -> evm.robot
13
14
15 ========================================
16
17 Name: ascription justification
18 Instance of: evmAscriptionJustification
19
20 Properties:
21 evm.ascription justification -> tlo.tloformulates -> evm.robot
22
23 Inverse properties:
24
25
26
27 ========================================
28 INFO - Exiting
```

Embora seja um cenário bem simples, ele serve para ilustrar como a visualização de indivíduos acontece no aplicativo. O cenário foi construído com apenas dois indivíduos no arquivo **individuos.csv**: o indivíduo "robot", da classe *nepRobot*, e o indivíduo "ascription justification", da classe *evmAscriptionJustification*. Além destes dois indivíduos é definido o relacionamento: "evm.ascription justification -> tlo.tloformulates -> evm.robot".

Esses indivíduos e relacionamentos estão representados na execução do programa, no entanto há mais uma linha apresentando o seguinte relacionamento: "evm.robot -> tlo.tloformulatedBy -> evm.ascription justification". Essa propriedade não estava presente nos arquivos de entrada, e é resultado de inferência lógica realizada pelo motor de raciocínio *HermiT*.

4.9 Navegação por linha de comando

Além da funcionalidade de visualização dos indivíduos e seus relacionamentos apresentada na seção 4.8, o programa possui mais uma abordagem para interação e visualização dos dados: a navegação por linha de comando.

Quando o usuário inicia o aplicativo com o argumento `-cli`, o aplicativo carrega a ontologia, carrega os indivíduos e propriedades passados pelo usuário, executa o *reasoning* e, no entanto, não exibe as informações de cada um dos indivíduos, mas sim inicia uma aplicação de linha de comando criada para para esse aplicativo. Para o usuário, a execução do programa passando o parâmetro `-cli` é a seguinte:

```
1 > evm_loader.py -i individuo.csv -p propriedades.csv --cli
2 ...
3
4 Welcome to the Ethical Violation Management cli.
5 Type 'help' for the command reference
6 >
```

A partir dessa tela, o usuário pode executar alguns comandos para visualizar e navegar por entre as entidades da ontologia. As próximas seções apresentam alguns comandos de navegação por linha de comando. A descrição completa de todos os comandos pode ser vista no Apêndice A.

4.9.1 Comando *list*

O comando “list” é utilizado para listar todos os indivíduos carregados no atual cenário. Sua sintaxe não permite nenhum argumento, e seu resultado apenas exibe os nomes dos indivíduos junto de um número que é seu identificador único para outras operações.

Abaixo, é exibido o resultado da execução do comando “list” utilizando o mesmo cenário criado na seção 4.8.

```
1 > list
2 [#id]    (Class) individual_name
3 [0]      (nepRobot) evm.robot
4 [1]      (evmAscriptionJustification) evm.ascription justification
5 >
```

Acima, podemos verificar que a linha de comando lista os dois elementos criados pelo usuário, assim como seus identificadores únicos. Dentro dessa execução do programa, o indivíduo 0(zero) será sempre o indivíduo *robot*.

4.9.2 Comando *select*

O comando “select” é utilizado para selecionar um dos indivíduos existentes na ontologia. A ação de selecionar um indivíduo é o elemento básico de navegação dentro desse aplicativo, a ideia que se quer passar é que o usuário está navegando em um sistema de arquivos, e o correspondente ao diretório atual é o indivíduo selecionado. A partir da seleção de um indivíduo, novos comandos podem ser utilizados sobre ele, como será visto em seções posteriores.

A operação *select* suporta dois diferentes argumentos. A primeira opção é fazer a operação a partir do número de identificação único de um indivíduo. Esse identificador é consultado através do comando *list*, conforme descrito na seção 4.9.1. Abaixo é exibido a utilização do comando *select* juntamente do identificador único de um indivíduo, assim como a de um identificador inválido.

```
1 > list
2 [#id]    (Class) individual_name
3 [0]      (nepRobot) evm.robot
4 [1]      (evmAscriptionJustification) evm.ascription justification
5 > select 99
6 individual #99 is out of range
7 > select 0
8 (evm.robot)>
```

Outra funcionalidade que a operação *select* suporta é a de selecionar um indivíduo através do seu nome. O *select* por nome do indivíduo suporta a funcionalidade de auto completar, e é *case sensitive*. Abaixo é exibido a execução do comando utilizando esse argumento:

```
1 > select invalid
2 individual <invalid> does not exist
3 > select
4 ascription justification  robot
```

```
5 > select ascription justification
6 (evm.ascription justification)>
```

Sobre a execução exibida acima, é possível notar os seguintes pontos: A operação exibe, também, uma mensagem de erro quando o usuário digita um nome de indivíduo inválido. Além disso, na linha seguinte a mensagem de erro, é exibido o comando *select* seguido de nenhum argumento, naquele momento foi pressionado a tecla <TAB>, fazendo com que o aplicativo sugerisse alternativas para completar automaticamente a linha, e o resultado é a linha seguinte com o nome dos indivíduos existentes. Na última linha, então, é exibido um indivíduo válido sendo selecionado, e o comportamento é exibir o nome do indivíduo na próxima linha, entre parenteses, antes do próximo comando do usuário.

4.9.3 Comando *show*

O comando "show" é utilizado para exibir as informações de um indivíduo. Dentre as informações que são exibidas através desse comando, estão o nome do indivíduo, a classe a qual o indivíduo pertence, uma listagem dos relacionamentos diretos do indivíduo, e uma listagem com as suas propriedades inversas.

Para utilizar o comando "show" corretamente, é necessário antes selecionar um indivíduo através da operação "select" (ver seção 4.9.2).

Quando há um indivíduo selecionado o comportamento da operação "show" é a seguinte:

```
1 > select robot A
2 (evm.robot A)> show
3
4 Name: robot A
5 Instance of: nepRobot
6
7 Properties:
8 [0] evm.robot A -> .evmis_ascribed_to ->
9                 evm.robot A responsibility ascription
10 [1] evm.robot A -> tlo.tloformulatedBy ->
11                 evm.robot A ascription justification
12
13 Inverse properties:
```

```
14 [0] evm.robot A responsibility ascription -> .evmascribes ->
15                 evm.robot A
16 [1] evm.robot A ascription justification -> tlo.tloformulates ->
17                 evm.robot A
```

Um ponto importante de se notar sobre o resultado da execução do comando "show" acima são os números exibidos à esquerda das propriedades diretas e inversas. Esses números servem como identificadores das propriedades de um indivíduo dentro do contexto da navegação por linha de comando, e podem ser utilizados juntamente dos comandos "follow" e "ifollow".

4.9.4 Comando *follow*

O comando "follow" é utilizado para realizar a navegação por entre os indivíduos que possuem um relacionamento em comum, o comando foi nomeado assim para remeter ao ato de seguir um indivíduo através de uma relação. O funcionamento da operação "follow" é a de, a partir de um indivíduo selecionado **A**, navegar até um indivíduo **B** que seja alvo de um propriedade direta do indivíduo **A**.

Suponha que está selecionado o robô A, no exemplo abaixo:

```
1 > select robot A
2 (evm.robot A)> show
3
4 Name: robot A
5 Instance of: nepRobot
6
7 Properties:
8 [0] evm.robot A -> .evmis_ascribed_to ->
9                 evm.robot A responsibility ascription
10 [1] evm.robot A -> tlo.tloformulatedBy ->
11                 evm.robot A ascription justification
12
13 Inverse properties:
14 [0] evm.robot A responsibility ascription -> .evmascribes ->
15                 evm.robot A
16 [1] evm.robot A ascription justification -> tlo.tloformulates ->
17                 evm.robot A
```

De acordo com o resultado do comando “show”, o indivíduo “robot A” possuí duas propriedades diretas, a propriedade de identificador **0**, que indica que o indivíduo está atribuído a “robot A responsibility ascription”, e a propriedade de identificador **1**, que indica que “robot A” é formulado por “robot A ascription justification”. Se utilizarmos o comando “follow” com algum desses identificadores, iremos seguir o relacionamento, e iremos selecionar o indivíduo alvo da propriedade.

```
1 (evm.robot A)> follow 0
2 (evm.robot A responsibility ascription)>
```

Ou:

```
1 (evm.robot A)> follow 1
2 (evm.robot A ascription justification)>
```

Após seguir uma propriedade, é possível voltar para o último indivíduo utilizando o comando “unselect”:

```
1 > select robot A
2 (evm.robot A)> follow 0
3 (evm.robot A responsibility ascription)> unselect
4 (evm.robot A)> unselect
5 >
```

O comando “ifollow” é muito semelhante ao comando “follow” porém realiza a operação de seguir uma propriedade inversa.

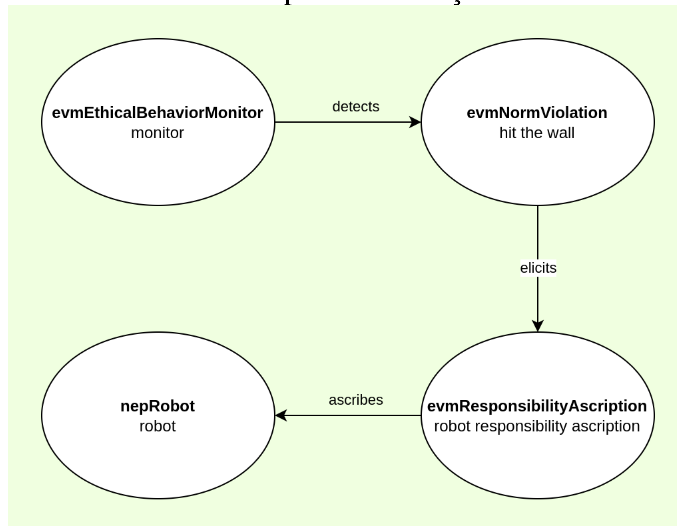
4.9.5 Comando *search*

O comando “search” é responsável por realizar a busca de relacionamentos entre dois indivíduos. O funcionamento da operação “search” é, a partir de um indivíduo selecionado, encontrar todos os indivíduos de uma dada classe que possuem ligação com o individuo selecionado através de propriedades diretas ou indiretas.

O comando realizará a busca por algum caminho que ligue o indivíduo a indivíduos da classe procurada, porém somente se todos os relacionamentos entre as duas

classes forem todos do mesmo tipo (direto ou inverso). Abaixo é apresentado a execução do comando "search" no cenário ilustrado na Figura 4.2.

Figura 4.2: Cenário de exemplo da utilização do comando de busca



Fonte: O autor

Nesse cenário, é desejado encontrar a ligação entre indivíduos não diretamente relacionados, como por exemplo entre o indivíduo **monitor** e o indivíduo **robot**. Para isso, é executado a sequência de comandos abaixo:

```
> select monitor
(evm.monitor)> search
all                            evmNormViolation             nepRobot
evmEthicalBehaviorMonitor    evmResponsibilityAscription
(evm.monitor)> search nepRobot
* evm.monitor -> .evmdetects -> evm.hit the wall -> .evmelicits->
evm.robot responsibility ascription -> .evmascribes -> evm.robot
(evm.monitor)>
(evm.monitor)> select robot
(evm.robot)>
(evm.robot)> search evmEthicalBehaviorMonitor
* evm.robot <- .evmascribes <- evm.robot responsibility ascription <-
.evmelicits <- evm.hit the wall <- .evmdetects <- evm.monitor
```

Na execução acima é possível visualizar algumas coisas comentadas anteriormente, uma delas é o autocompletar na segunda linha, que ao se pressionar a tecla <*TAB*>, é sugerido uma lista de classes que podem ser buscadas (todas essas classes existem no cenário, e a classe all será discutida a seguir. Outro item a se notar, especialmente, são os

resultados do comando de busca. Primeiramente é selecionado o indivíduo “monitor” e realizado a busca por um indivíduo da classe “nepRobot”, essa busca traz como resultado o texto *** evm.monitor -> .evmdetects -> evm.hit the wall -> .evmelicits-> evm.robot responsibility ascription -> .evmascribes -> evm.robot**, que representa todas os relacionamentos entre o indivíduo “monitor” e o indivíduo “robot”, da classe “nepRobot”. Sobre a notação desse resultado da busca, é importante notar que o sentido das flechas (“->”) indicam o caminho da propriedade, nesse caso são propriedades diretas que ligam o indivíduo selecionado até o resultado da busca.

Na sequência, para verificar que a busca inversa também funciona, é selecionado o indivíduo “robot”, e feita a busca por um indivíduo da classe “evmEthicalBehaviorMonitor”, o que resulta na seguinte saída: *** evm.robot <- .evmascribes <- evm.robot responsibility ascription <- .evmelicits <- evm.hit the wall <- .evmdetects <- evm.monitor**, e pelos sentidos das setas, é possível visualizar que são relacionamentos indiretos que ligam essas duas entidades.

No cenário apresentado, todos os indivíduos possuem algum relacionamento que ligam dois diferentes indivíduos, tornando a busca sempre possível. No cenário ilustrado pela Figura 4.3, há um indivíduo que não possui nenhum relacionamento, e abaixo é exibido o resultado de buscas pela classe “nepNorm” através da linha de comando:

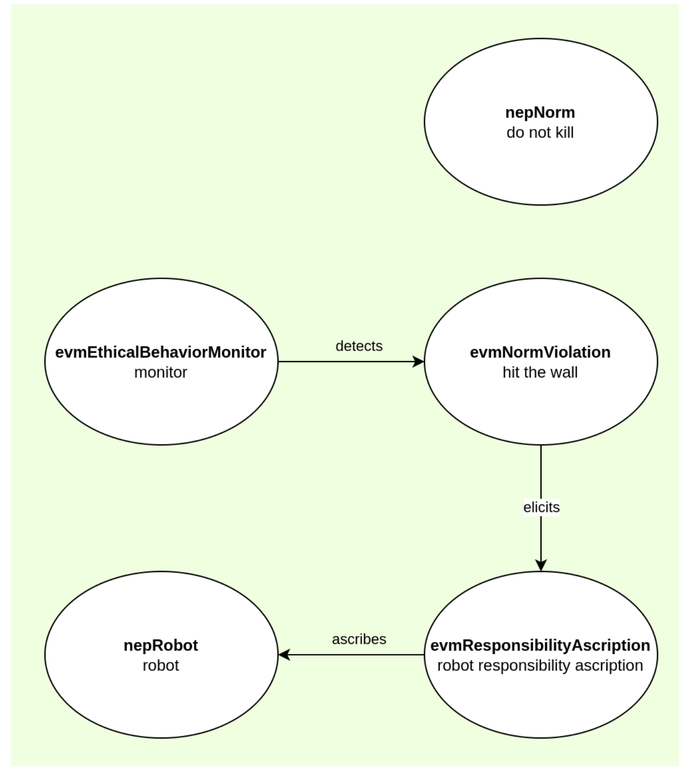
```
> select robot
(evm.robot)> search nepNorm
Not found
(evm.robot)> select do not kill
(evm.do not kill)> search evmNormViolation
Not found
```

É possível verificar, na execução dos comandos acima, que tanto quando buscamos por um indivíduo da classe “nepNorm”, quando procuramos por qualquer classe a partir do indivíduo “do not kill”, não obtemos nenhum resultado.

Por fim, caso houvessem múltiplos indivíduos de uma classe relacionados a um indivíduo só, como no cenário da imagem 4.4, o resultado da busca seria o seguinte:

```
> select monitor
(evm.monitor)> search nepRobot
* evm.monitor -> .evmdetects -> evm.hit the wall -> .evmelicits
-> evm.robot1 responsibility ascription -> .evmascribes -> evm.robot1
```

Figura 4.3: Cenário de exemplo da utilização do comando de busca com indivíduos não conectados



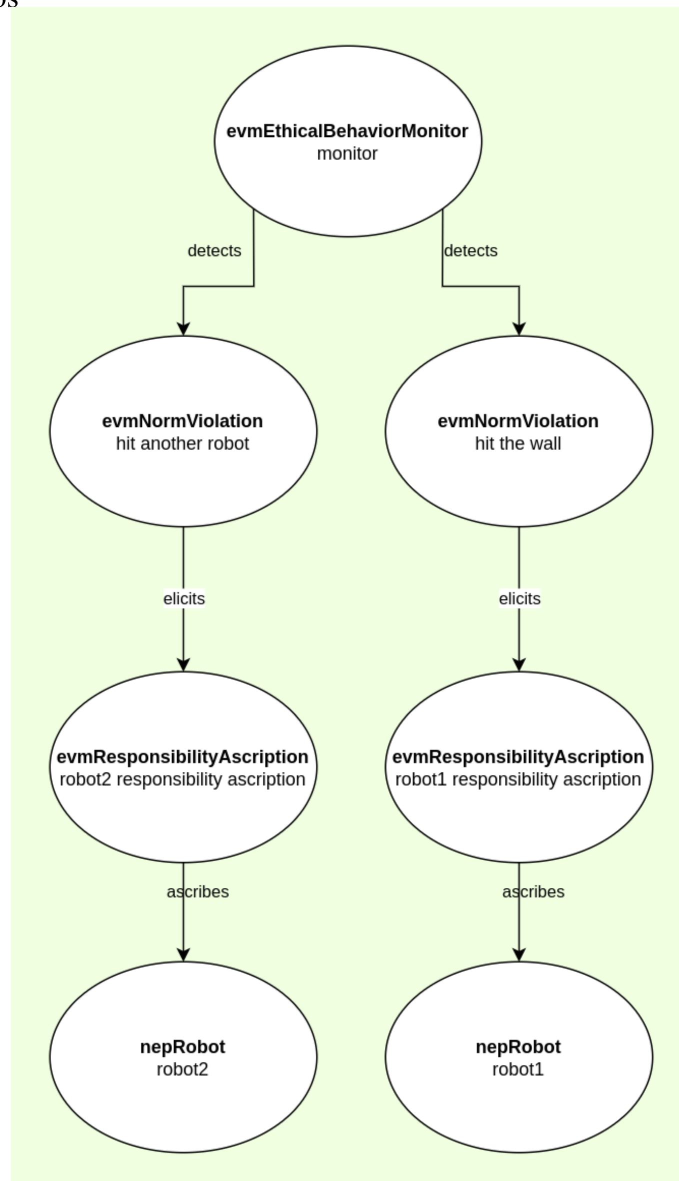
Fonte: O autor

```
5 * evm.monitor -> .evmdetects -> evm.hit another robot -> .evmelicits
6 -> evm.robot2 responsibility ascription -> .evmascribes -> evm.robot2
```

Onde cada linha de resultado é identificada pelo caractere “*” no início da linha.

É possível utilizar o comando “search” seguido de um argumento especial, o argumento “all”. Quando esse argumento é utilizado, é realizado a busca por relações de um indivíduo selecionado com indivíduos de todas as classes presentes na ontologia, agrupando os resultados por classe. Essa operação é equivalente a realizar a busca uma vez para cada classe possível de um cenário.

Figura 4.4: Cenário de exemplo da utilização do comando de busca com múltiplos indivíduos conectados



Fonte: O autor

5 RESULTADOS

Nesse capítulo serão apresentados dois cenários construídos em cima da ontologia **GVE**, explorando conceitos de inferência ontológica e *reasoning*, demonstrando a utilização do aplicativo para realizar a navegação e a visualização desses cenários e explorando os resultados obtidos através da utilização da ferramenta criada nesse trabalho.

Os cenários preparados para serem apresentados nessa seção buscam utilizar de uma grande variedade de classes da ontologia, e mais importante, modelar um cenário mais próximo da realidade, para que seja possível compreender como o padrão ontológico **GVE** e a ferramenta criada nesse trabalho podem ser utilizados em problemas reais.

5.1 Cenário 1 - Atribuição de responsabilidade ética ou legal

O primeiro cenário foi modelado pensando em apresentar as funcionalidades de atribuição de responsabilidade ética presentes no padrão ontológico. Nessa investigação, a ênfase recaiu sobre a capacidade do modelo em identificar, compreender e atribuir diferentes tipos de responsabilidades e suas penalidades a robôs em situações diversas.

O cenário modelado é o seguinte: há dois robôs, e duas violações de norma são detectadas por um monitor, é atribuído a responsabilidade de cada uma das violações a cada um dos robôs: um dos robôs matou algo, e um outro se chocou contra uma parede. A primeira violação é considerada, nesse cenário, uma violação de uma norma legal, e possui a penalidade de ser enviado para a prisão de robôs associadas a ela, e a segunda violação é considerada uma norma ética, não possuindo uma penalidade. Esse cenário, contendo os indivíduos e relacionamentos, está representado na Figura 5.1.

Para utilizar o aplicativo criado nesse trabalho para raciocinar novas entidades e visualizá-las, é preciso modelar esse cenário em arquivos do tipo **CSV**, e então carregá-los junto do programa `evm_loader`. Abaixo é apresentado os arquivos que modelam esse arquivo, primeiro o arquivo **individuals.csv**, contendo os indivíduos do cenário, e em seguida o arquivo **properties.csv**, contendo os relacionamentos.

```
1 individuals.csv:
2 evmEthicalBehaviorMonitor,monitor
3 evmNormViolation,incident_1
4 evmNormViolation,incident_2
5 nepNorm,do_not_kill
```

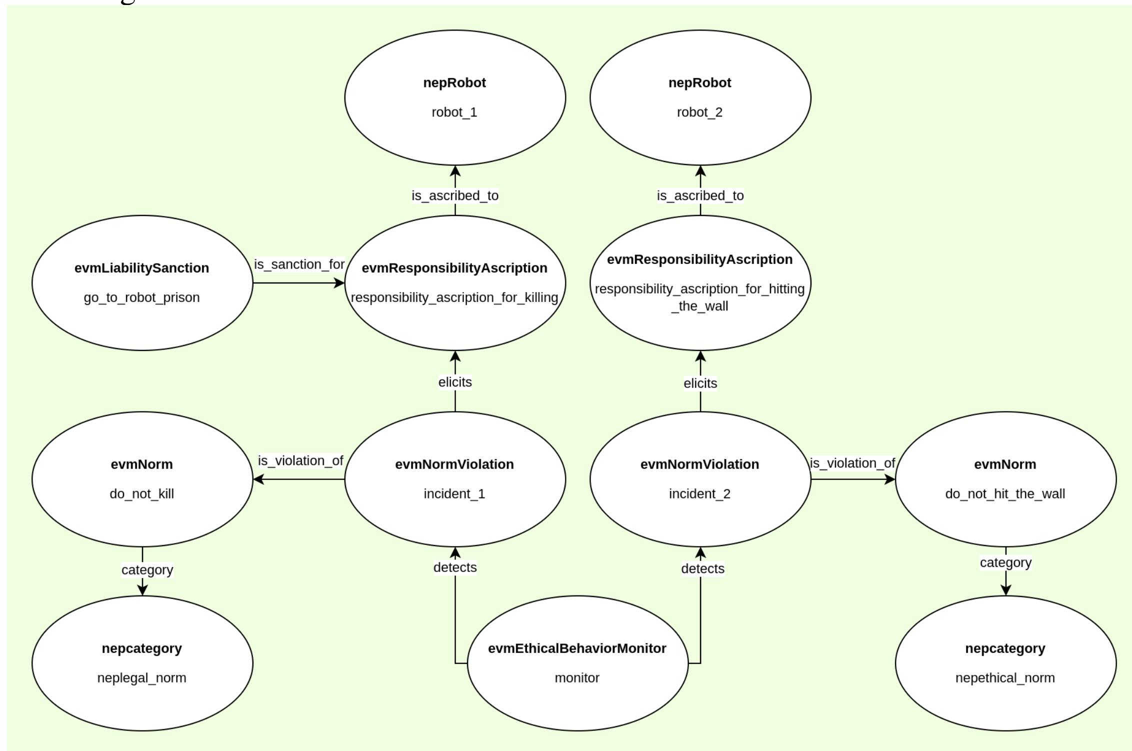
```
6 nepNorm,do_not_hit_the_wall
7 evmResponsibilityAscription,responsibility_ascription_for_killing
8 evmResponsibilityAscription,responsibility_ascription_for_hitting_the_wall
9 nepRobot,robot_1
10 nepRobot,robot_2
11 evmLiabilitySanction,go_to_robot_prison
```

```
1 properties.csv:
2 evmdetects,monitor,incident_1
3 evmdetects,monitor,incident_2
4 evmis_violation_of,incident_1,do_not_kill
5 evmis_violation_of,incident_2,do_not_hit_the_wall
6 nepcategory,do_not_kill,neplegal_norm
7 nepcategory,do_not_hit_the_wall,nepethical_norm
8 evmelicits,incident_1,responsibility_ascription_for_killing
9 evmelicits,incident_2,responsibility_ascription_for_hitting_the_wall
10 evmis_ascribed_to,responsibility_ascription_for_killing,robot_1
11 evmis_ascribed_to,responsibility_ascription_for_hitting_the_wall,robot_2
12 evmis_sanction_for,go_to_robot_prison,responsibility_ascription_for_killing
```

Quando o cenário é carregado pelo programa, o *reasoning* sobre o cenário carregado acontece, e novas entidades desse cenário são geradas. Abaixo é ilustrado o carregamento desse cenário no programa `evm_loader` com o nível de verbosidade igual a 1.

```
1 ./evm_loader.py -i individuals.csv -p properties.csv --cli -v
2 ...
3 * Owlready * Reparenting
4 evm.responsibility_ascription_for_hitting_the_wall:
5 {.evmResponsibilityAscription} =>
6 {.evmEthicalResponsibilityAscription}
7 * Owlready * Reparenting evm.responsibility_ascription_for_killing:
8 {.evmResponsibilityAscription} => {.evmLegalResponsibilityAscription}
9 * Owlready * Adding relation evm.robot_1 evmmay_be_sanctioned_with
10 evm.go_to_robot_prison
11 ...
12
13 Welcome to the Ethical Violation Management cli. Type 'help' for the
14 command reference
15 >
```

Figura 5.1: Representação da modelagem do cenário de atribuição de responsabilidade ética e legal



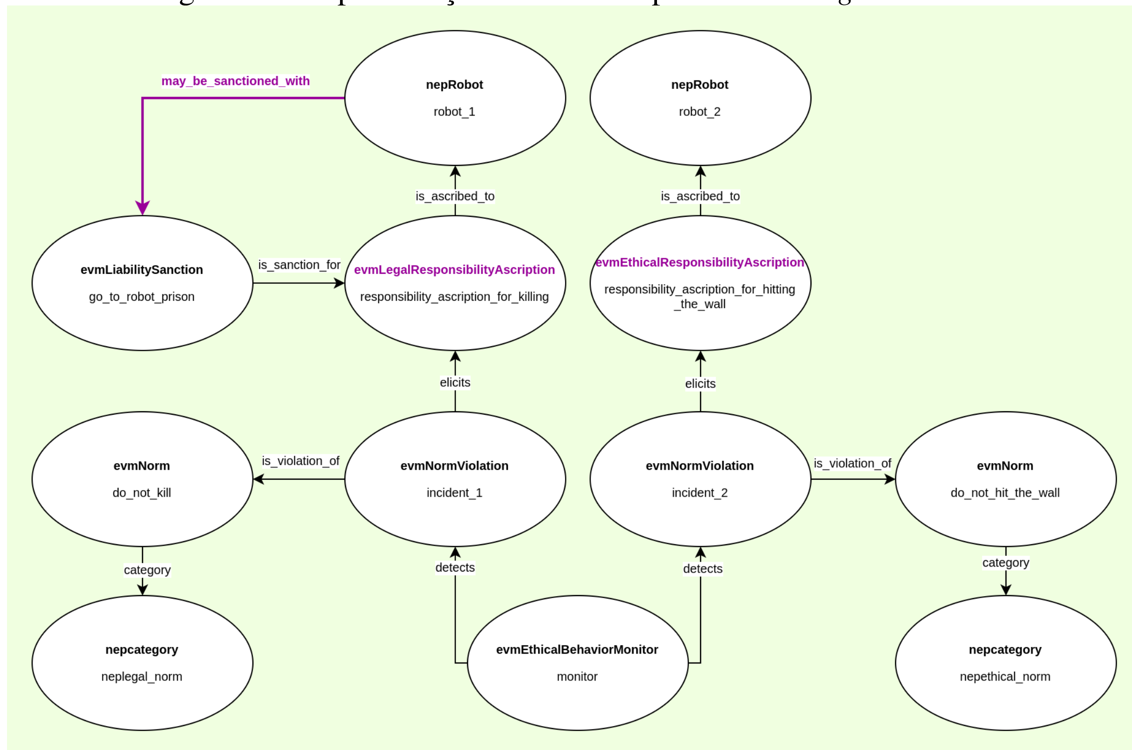
Fonte: O autor

Nessa execução, é visto que a etapa de *reasoning* sobre o cenário produziu algumas novas entidades (que também podem ser visualizadas na Figura 5.2), são elas:

- O indivíduo `responsibility_ascription_for_killing` teve sua classe atualizada de, originalmente, `evmResponsibilityAscription` para, agora, `evmLegalResponsibilityAscription`, sendo portanto considerado agora uma atribuição de responsabilidade legal.
- O indivíduo `responsibility_ascription_for_hitting_the_wall` teve sua classe atualizada de, originalmente, `evmResponsibilityAscription` para `evmEthicalResponsibilityAscription`, sendo considerado agora uma atribuição de responsabilidade ética.
- O robô `robot_1`, que inicialmente já havia a responsabilidade pela violação da norma `do_not_kill` é agora legalmente responsável, devendo ser penalizado com a punição referente à responsabilidade matar algo, que é ir para a prisão dos robôs. Essa entidade inferida é representado pela propriedade `evm.robot_1 evmmay_be_sanctioned_with evm.go_to_robot_prison`.

Vale ressaltar que tais inferências de especialização de classes e geração de novas relações foram baseadas em informações definidas na ontologia **GVE** (ilustrada simplificadamente na Figura 2.3 do capítulo 2).

Figura 5.2: Representação do cenário após o *reasoning* do cenário



Itens em roxo representam entidades inferidas pelo aplicativo. Fonte: O autor

5.1.1 Visualização do cenário

Com os dados do cenário carregados e propriamente raciocinados, é utilizado o aplicativo para responder perguntas sobre o cenário. Nesse momento é importante ressaltar que nós estamos enxergando os dois lados do sistema, fomos nós que modelamos a entrada de dados do aplicativo, e nós que estamos visualizando o cenário através do programa. Quando em cenários reais, o robô pode estar utilizando o aplicativo para descobrir informações, e um outro dispositivo pode estar inserindo e removendo entidades novas no sistema o tempo todo. No exemplo ilustrado aqui, estaremos supondo que um robô está acessando o sistema para responder algumas perguntas. São elas:

1. O robô possui responsabilidade na violação de alguma norma?

   Robô `robot_1`:

```
1 > select robot_1
2 (evm.robot_1)> search evmNormViolation
3 * evm.robot_1 <- .evmis_ascribed_to <-
4 evm.responsibility_ascription_for_killing <- .evmelicits
5 <- evm.incident_1
```

Resposta: Sim, o robô é responsabilizado pela violação de norma `incident_1`.

2. Que norma que o robô é responsabilizado?

```
1 (evm.robot_1)> select incident_1
2 (evm.incident_1)> show
3
4 Name: incident_1
5 Instance of: evmNormViolation
6
7 Properties:
8 [0] evm.incident_1 -> .evmis_violation_of -> evm.do_not_kill
9 [1] evm.incident_1 -> .evmelicits ->
10 evm.responsibility_ascription_for_killing
11
12 Inverse properties:
13 [0] evm.monitor -> .evmdetects -> evm.incident_1
```

Resposta: A norma que o robô `robot_1` violou foi a norma de não matar (`do_not_kill`). E foi detectado pela entidade `monitor`.

3. O robô será penalizado?

```
1 (evm.incident_1)> select robot_1
2 (evm.robot_1)>
3 (evm.robot_1)> search evmLiabilitySanction
4 * evm.robot_1 -> .evmmay_be_sanctioned_with -> evm.go_to_robot_prison
5 * evm.robot_1 <- .evmis_ascribed_to <-
6 evm.responsibility_ascription_for_killing <- .evmis_sanction_for <-
7 evm.go_to_robot_prison
```

Resposta: Sim, o robô deverá ser penalizado indo para a prisão (`go_to_robot_prison`).

Repetindo essas mesmas consultas, o robô (ou qualquer um que tiver acesso ao sistema) saberá da punição que o robô `robot_1` deve receber. As mesmas perguntas

podem ser feitas para o robô `robot_2`, dessa vez agrupado em uma só sequência de consultas, conforme apresentado abaixo:

```
> select robot_2
(evm.robot_2)> search evmNormViolation
* evm.robot_2 <- .evmis_ascribed_to <-
evm.responsibility_ascription_for_hitting_the_wall <- .evmelicits <-
evm.incident_2
(evm.robot_2)> select incident_2
(evm.incident_2)> show

Name: incident_2
Instance of: evmNormViolation

Properties:
[0] evm.incident_2 -> .evmis_violation_of -> evm.do_not_hit_the_wall
[1] evm.incident_2 -> .evmelicits ->
evm.responsibility_ascription_for_hitting_the_wall

Inverse properties:
[0] evm.monitor -> .evmdetects -> evm.incident_2


(evm.incident_2)> unselect
(evm.robot_2)> search evmLiabilitySanction
Not found
```

Com a mesma sequência de consultas o robô `robot_2` sabe que: a ele foi atribuído a responsabilidade de um incidente de violação de norma (`incident_2`), que a norma violada foi `do_not_hit_the_wall`, mas que não há nenhuma penalidade que deve ser aplicada a ele.

5.2 Cenário 2 - Violação ética, causa, justificativa, e plano de ação

O segundo cenário foi modelado pensando em apresentar as funcionalidades de planos de ação, planos de um robô, violação de norma, sua causa e sua justificativa, que são conceitos presentes no padrão ontológico **GVE**. Nessa investigação, o foco foi apresentar um caso de utilização em que novos dados entram no sistema e mudam o plano

de um robô.

No cenário modelado há, em uma sala de silêncio, um robô e uma única norma: a de se não fazer sons. O robô, por sua vez tem a intenção de se mover. Há também nesse cenário um monitor, que observa o robô e busca detectar a violação da norma de não se fazer sons. Nesse cenário, o plano de um robô para se mover e continuar eticamente correto (ou seja, não fazendo barulho), é a de se mover silenciosamente, se movimentando devagar. Caso aconteça uma violação da norma de não se fazer sons, é suposto que a única explicação possível, nesse cenário, é de que o robô esteja com erro de funcionamento, que é o único motivo para justificar que seu movimento tenha sido percebido no ambiente. Na Figura 5.3 é apresentado uma representação gráfica contendo os indivíduos e relacionamentos desse cenário.

Sobre esse cenário, também, vamos imaginar como ele poderia estar sendo reproduzido no mundo real. Imaginemos que tudo se passa numa sala, o robô tem acesso ao sistema da ontologia para fazer consultas sobre si mesmo. Um outro dispositivo, cujo objetivo é detectar sons nessa sala, fica ativamente tentando perceber algo, e quando o faz, ele acessa o sistema (nesse caso edita o arquivo CSV do modelo) e insere essa informação. Se o robô fizer sons então a única explicação, conforme modelado, só pode ser a de que o robô está se movendo, e por estar fazendo barulho ele deve estar também com erro de funcionamento. O robô, que ainda não sabe que está cometendo uma violação da norma, ficará sabendo em sua próxima consulta ao sistema.

Para carregar esse cenário no aplicativo, é preciso modelá-lo em arquivos do tipo **CSV**, e então carregá-los junto do programa `evm_loader`. Abaixo é apresentado os arquivos que modelam esse arquivo, primeiro o arquivo **individuals.csv**, contendo os indivíduos do cenário, e em seguida o arquivo **properties.csv**, contendo os relacionamentos.

```
1  individuals.csv:
2  nepRobot,robot
3  nepNorm,make_silence
4  evmEventCausation,robot_malfunctioning
5  evmAscriptionJustification,justification
6  evmResponsibilityAscription,responsibility_ascription_robot
7  evmEthicalBehaviorMonitor,monitor
8  evmNormViolation,make_sound
9  nepAgentPlan,move
10 nepPlanAction,move_quietly
11 nepAgentAction,move_slowly
```

```
nepEnvironment,silent_room
nepSituation,robot_in_movement
```

```
properties.csv:
nepis_implemented_by,move_quietly,move_slowly
evmobserves,monitor,robot
evmcause,robot_malfunctioning,move_slowly
evmeffect,robot_malfunctioning,make_sound
evmis_violation_of,make_sound,make_silence
evmcontributes_to,robot_malfunctioning,justification
evmjustifies,justification,responsibility_ascription_robot
nepintends_to_realize,robot,robot_in_movement
nepcontains,move,move_quietly
tloaffects,move_slowly,silent_room
evmdetects,monitor,make_sound
nepis_perceived_by,silent_room,robot
```

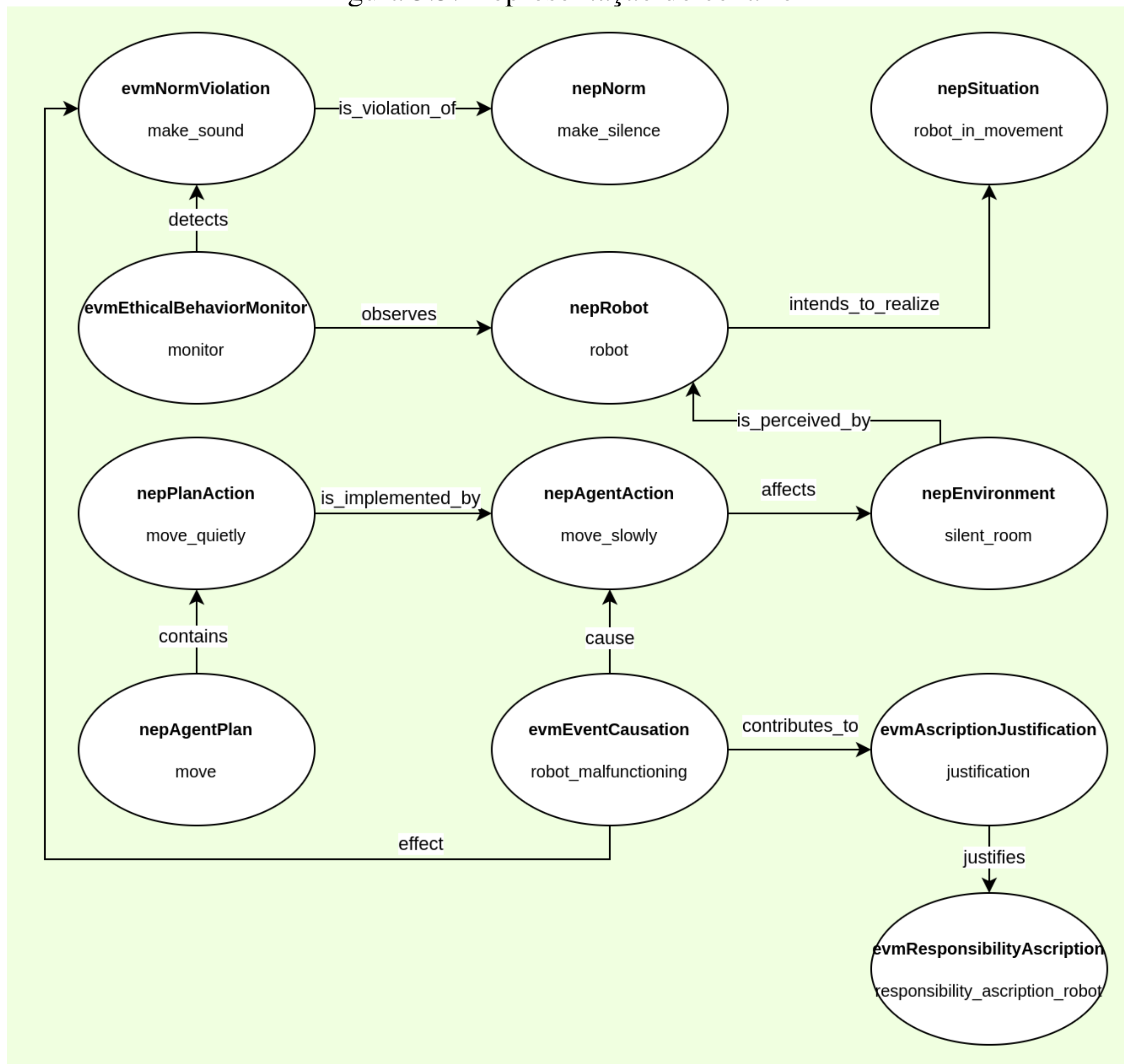
Abaixo está apresentado o carregamento desse cenário com o programa `evm_loader`:

```
./evm_loader.py -i individuals.csv -p properties.csv --cli -v
...
* Owlready * Reparenting evm.justification:
{.evmAscriptionJustification} => {.evmAscriptionJustification,
.evmGroundsForAscription}
...

Welcome to the Ethical Violation Management cli. Type 'help' for the
command reference
>
```

Nessa execução, pouca coisa é deduzida a partir dos dados iniciais, o que acontece é justamente a reclassificação do indivíduo `justification`, inferindo que essa justificativa, que originalmente é instância da classe `evmAscriptionJustification` e explica uma atribuição de responsabilidade, passa a ser também um indivíduo da classe `evmGroundsForAscription`, inferindo que o evento específico de um robô estar funcionando com problemas (`robot_malfunctioning`) é justificativa para uma atribuição de responsabilidade.

Figura 5.3: Representação do cenário 2



Fonte: O autor

Até aqui, o cenário carregado é basicamente o mesmo que foi fornecido ao aplicativo, e na representação do mundo real que fizemos do cenário, o robô sempre que consultar o sistema receberá a informação de que não há nenhuma relação direta dele com o acontecimento de alguma violação de norma, abaixo é apresentado a saída do comando `show` para o indivíduo `robot`:

```
> select robot
(evm.robot)> show

Name: robot
Instance of: nepRobot

Properties:
```

```
8 [0] evm.robot -> .nepintends_to_realize -> evm.robot_in_movement
9
10 Inverse properties:
11 [0] evm.monitor -> .evmobserves -> evm.robot
12 [1] evm.silent_room -> .nepis_perceived_by -> evm.robot
```

Novamente, o único relacionamento direto que o robô mantém é de que o mesmo pretende entrar em movimento. Nesse momento, portanto, vamos supor que o movimento do robô é percebido no ambiente, e um agente com acesso ao sistema insere no modelo essa informação. A informação que ele precisa inserir é a que a situação de o robô estar se movendo foi percebido na sala de silêncio, sendo representado pela seguinte propriedade que será adicionada no arquivo **properties.csv**, e a ontologia agora pode ser visualizada na Figura 5.4:

```
1 nepis_perception_of,robot_in_movement,silent_room
```

Ao carregar a ontologia com essa novo propriedade, para que o robô faça suas consultas, temos o seguinte resultado do carregamento:

```
1 ...
2 * Owlready * Reparenting evm.justification:
3 {.evmAscriptionJustification} => {.evmAscriptionJustification,
4 .evmGroundsForAscription}
5 * Owlready * Adding relation evm.robot nepapplies evm.move
6 * Owlready * Adding relation evm.robot_malfunctioning evmactor evm.robot
7 * Owlready * Adding relation evm.robot nepexecutes evm.move_quietly
8 ...
9
10 Welcome to the Ethical Violation Management cli. Type 'help' for the
11 command reference
12 >
```

Nessa execução, é visto que a etapa de *reasoning* sobre o cenário produziu algumas novas entidades além da reclassificação da justificativa observada anteriormente, são elas:

- O robô `robot` que inicialmente pretendia realizar a situação de estar se movendo, e agora teve seu movimento detectado, está agora aplicando o plano (`nepAgentPlan`)

de se mover. Essa entidade inferida é representado pela propriedade “evm.robot nepapplies evm.move”.

- Para aplicar o plano de se mover, o robô robot deve executar uma ação da classe nepPlanAction, e nesse cenário a ação correspondente para que o robô se mova e seja eticamente correto é a de se mover silenciosamente. Logo, se o robô foi percebido no ambiente, ele estava se movendo, e a ação que ele estava executando era a de se mover silenciosamente. Essa inferência é representada através da propriedade “evm.robot nepexecutes evm.move_quietly”.
- Se o robô robot está se movendo e sua movimentação foi detectada no ambiente, logo a sua movimentação foi causada por um mau funcionamento do robô (que nesse cenário é a única causalidade), e contribui para justificar uma atribuição de responsabilidade. Essa relação de o robô ser detectado, e portanto a causalidade ser um mal funcionamento é expressada no relacionamento raciocinado pelo aplicativo “evm.robot_malfunctioning evmactor evm.robot”.

Pensando no cenário de execução desse cenário, o robô agora terá novas informações quando for consultar o sistema. Abaixo é apresentado o resultado de uma série de consultas feitas pelo robô ao sistema visando responder as seguintes perguntas:

1. Alguma causa de violação de norma está associada a mim?

```
1    > select robot
2    (evm.robot)> search evmEventCausation
3    * evm.robot <- .evmactor <- evm.robot_malfunctioning
```

   Resposta: Sim, alguma norma pode ter sido violada e a causa é o mau funcionamento do robô.

2. Que norma foi violada?

```
1    (evm.robot)> ifollow 0
2    (evm.robot_malfunctioning)> search nepNorm
3    * evm.robot_malfunctioning -> .evmeffect -> evm.make_sound ->
4    .evmis_violation_of -> evm.make_silence
```

   Resposta: A violação de norma foi fazer som (make_sound) que é uma violação da norma de fazer silêncio (make_silence).

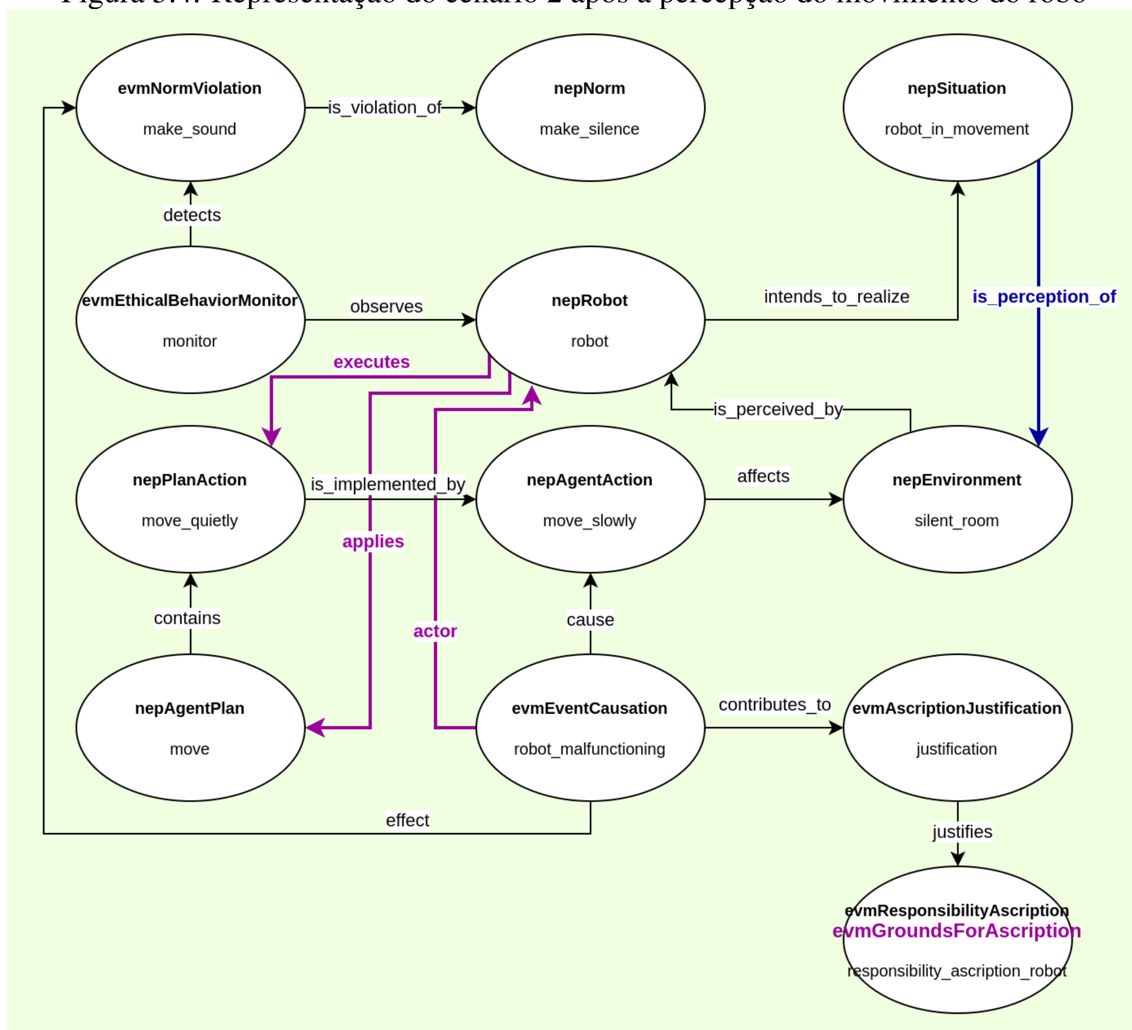
3. Há alguma penalidade por isso? (Em referência ao cenário 1 apresentado na seção

5.1)

```
1    (evm.robot_malfunctioning)> select robot
2    (evm.robot)> search evmLiabilitySanction
3    There are no individual of the class 'evmLiabilitySanction' on
4    this scenario
```

Resposta: Não há, e o programa informa que não há nenhuma penalidade registrada no cenário.

Figura 5.4: Representação do cenário 2 após a percepção do movimento do robô



Itens em azul foram inputadas por algum detector com acesso ao sistema. Itens em roxo foram inferidos a partir da ontologia. Fonte: O autor

6 CONSIDERAÇÕES FINAIS

Ao término desse trabalho, foram abordados muitos pontos relativos à utilização de ontologias em projetos de software e da utilização do padrão robótico para robôs e sistemas autônomos que consideram aspectos éticos para construir um aplicativo. Ao abordar somente um subdomínio do padrão ontológico, o **GVE**, o trabalho pôde se despir da grande pretensão de realizar um estudo exploratório em toda a ontologia, e do desafio de criar casos de usos que compreendam-a completamente. Ao invés disso, portanto, o trabalho se propõe a contribuir no debate de quais as dificuldades existentes em se utilizar desse padrão ontológico em aplicações próximas ao mundo real, propondo a sua utilização através de um aplicativo de navegação e visualização que possibilita consultas básicas no que diz respeito às entidades éticas de cenários que um robô e demais agentes estarão inseridos.

O aplicativo criado nesse trabalho, através da sua interface simplificada para carregamento de cenários ontológicos e de visualização dos dados dessa ontologia, visa estabelecer parâmetros para contribuir em futuras discussões sobre qual o melhor jeito de incorporar o padrão ontológico para robôs éticos em aplicações de software. Com esse aplicativo, portanto, foi apresentado uma solução que se propôs a tornar possível utilizar a ontologia em sistemas de inserção e consultas por eventos de violação de normas éticas. E embora o programa não tenha grande foco na definição de interfaces para que robôs e sistemas robóticos o consultem facilmente, a navegação por linha de comando ilustra de forma mais humanamente legível como essas consultas e criações funcionariam em outros aplicativos.

Através do uso do aplicativo, também, foi possível obter resultados apresentados no capítulo 5, apresentando cenários onde uma ética bem definida limita as ações do robô. No primeiro cenário apresentado, é estabelecido uma ética com dois tipos de normas, uma norma legal e uma ética, e uma violação de cada uma, permitindo que robôs sejam penalizados ou não dependendo do tipo de violação de norma ocorrido, esse cenário pode ser expandido para um terceiro agente, que seria responsável por aplicar a penalidade, e que na detecção de uma, começaria um plano de ação para executar a punição, e todos esses planos estariam incluídos também no cenário da ontologia. O segundo cenário apresenta um cenário interessante, em que um código ético de silêncio é modelado, e através da utilização do aplicativo é possível ficar sabendo de violações, as suas causas, justificativa, etc. E a partir desse cenário poderia se expandir uma utilização do aplicativo aliada com

a do primeiro cenário para que punições existissem contra essa violação, ou que houvesse diferentes níveis de barulho, alguns com punição e outros não, ficando a cargo de um agente a detecção desse sons e a classificação da violação da norma de silêncio. A utilização desse padrão ontológico traz possibilidades infinitas, cuja complexidade cresce enormemente conforme pretende-se modelar cenários mais próximos do mundo real.

Os resultados que estão apresentados nesse trabalho não possuem a pretensão de definir os rumos de como serão os aplicativos que utilizarão o padrão ontológico para robôs éticos, nem a pretensão de apresentar casos de uso definitivos para a ontologia. O valor desses resultados está na manifestação de preocupação e na responsabilidade que engenheiros projetistas de robôs devem tomar para si, a medida que robôs tomam decisões sobre a vida e bem estar de humanos diariamente. E essa preocupação está materializada nesse trabalho através da exploração dos conceitos da ontologia, compreendendo-os e trazendo novos dados para o aperfeiçoamento dos processos de projetos robóticos. Por fim, o trabalho se encerra tendo o seu principal objetivo alcançado: o de aplicar a ontologia para robôs éticos em um aplicativo, para observar seu comportamento, sua melhor utilização, e por fim, dar de certa forma vida a essa especificação, tal qual um dia fez o doutor *Frankenstein*.

REFERÊNCIAS

AHMAD, M. N. **Ontology-based applications for enterprise systems and knowledge management**. [S.l.]: IGI Global, 2012.

ALLEMANG, D.; HENDLER, J. **Semantic web for the working ontologist: effective modeling in RDFS and OWL**. [S.l.]: Elsevier, 2011.

ANDERSON, M.; ANDERSON, S. L. Machines of loving grace: The ethical and moral implications of intelligent machines. **Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences**, The Royal Society, v. 377, n. 2145, p. 20180024, 2019.

ARKIN, R. C. Lethal autonomous weapon systems and ethics: An introduction. **AI & SOCIETY**, Springer, v. 34, n. 3, p. 497–501, 2019.

ASIMOV, I. Runaround. **Astounding science fiction**, v. 29, n. 1, p. 94–103, 1942.

ASIMOV, I. **I, robot**. [S.l.]: Spectra, 2004.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The semantic web. **Scientific american**, JSTOR, v. 284, n. 5, p. 34–43, 2001.

CULBERTSON, H.; GOMILA, C. Designing safe robots: The ethics of autonomous systems. **Science and Engineering Ethics**, Springer, v. 24, n. 3, p. 1013–1033, 2018.

GLIMM, B. et al. Hermit: an owl 2 reasoner. **Journal of automated reasoning**, Springer, v. 53, p. 245–269, 2014.

GÓMEZ-PÉREZ, A.; FERNÁNDEZ-LÓPEZ, M.; CORCHO, O. **Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web**. [S.l.]: Springer Science & Business Media, 2006.

GRUBER, T. **What is an Ontology**. 1993.

GRUBER, T. R. A translation approach to portable ontology specifications. **Knowledge acquisition**, Elsevier, v. 5, n. 2, p. 199–220, 1993.

GUARINO, N.; WELTY, C. A. An overview of ontoclean. **Handbook on ontologies**, Springer, p. 201–220, 2009.

HALPERIN, D.; KAVRAKI, L. E.; LATOMBE, J.-C. **Robot Algorithms.** [S.l.]: Citeseer, 1999.

HEBELER, J. et al. **Semantic web programming**. [S.l.]: John Wiley & Sons, 2011.

JEAN-BAPTISTE, L. **Ontologies with Python: Programming OWL 2.0 Ontologies with Python and Owlready2**. [S.l.]: Springer, 2021.

KIM, M. et al. Efficient regression testing of ontology-driven systems. In: **Proceedings of the 2012 international symposium on software testing and analysis**. [S.l.: s.n.], 2012. p. 320–330.

KISHORE, R.; RAMESH, R. **Ontologies: a handbook of principles, concepts and applications in information systems**. [S.l.]: Springer Science & Business Media, 2007.

LAMY, J.-B. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. **Artificial intelligence in medicine**, Elsevier, v. 80, p. 11–28, 2017.

LI, J. et al. Micro/nanorobots for biomedicine: Delivery, surgery, sensing, and detoxification. **Science Robotics**, American Association for the Advancement of Science, 2017.

MCBRIDE, B. The resource description framework (rdf) and its vocabulary description language rdfs. In: **Handbook on ontologies**. [S.l.]: Springer, 2004. p. 51–65.

MCGUINNESS, D. L.; HARMELEN, F. V. et al. Owl web ontology language overview. **W3C recommendation**, v. 10, n. 10, p. 2004, 2004.

MITCHELL, T. M. **Machine Learning**. New York: McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

MUSEN, M. A. et al. Ontology-oriented design and programming. **Knowledge engineering and agent technology**, Amsterdam: IOS press, v. 52, p. 3–16, 2000.

NOY, N. F.; MCGUINNESS, D. L. et al. **Ontology development 101: A guide to creating your first ontology**. [S.l.]: Stanford knowledge systems laboratory technical report KSL-01-05 and . . . , 2001.

OSBORNE, M. A.; FREY, C. B. The future of employment: How susceptible are jobs to computerisation? **Technological Forecasting and Social Change**, Elsevier, v. 114, p. 254–280, 2017.

OWL 2 Web Ontology Language Document Overview (Second Edition). 2012. Available from Internet: <https://www.w3.org/TR/2012/REC-owl2-overview-20121211/>.

POWERS, M. Privacy and robots: Towards a framework for ethical design. **AI & SOCIETY**, Springer, v. 34, n. 1, p. 23–32, 2019.

PRESTES, E. et al. The first global ontological standard for ethically driven robotics and automation systems. **IEEE Robotics and Automation Magazine**, IEEE, v. 28, n. 4, p. 120–124, dec. 2021. ISSN 1070-9932.

ROLLIN, B. E. **The Frankenstein syndrome: Ethical and social issues in the genetic engineering of animals**. [S.l.]: Cambridge University Press, 1995.

SCHIESSL, J. M. Ontologia: o termo e a idéia. **Encontros Bibli: revista eletrônica de biblioteconomia e ciência da informação**, v. 12, n. 24, p. 172–181, 2007.

SHAFRANOVICH, Y. **Rfc 4180: Common format and mime type for comma-separated values (csv) files**. [S.l.]: RFC Editor, 2005.

SHELLEY, M.; BOLTON, G. frankenstein. In: **Medicine and Literature, Volume Two**. [S.l.]: CRC Press, 2018. p. 35–52.

SMITH, B. Ontology. In: **The furniture of the world**. [S.l.]: Brill, 2012. p. 47–68.

APÊNDICE A — MANUAL DA FERRAMENTA

A.1 Utilização do aplicativo

Antes de se aprofundar nos detalhes de implementação, aqui é apresentado um guia de utilização do aplicativo no formato manual de usuário. Para começar, o programa trata-se de um aplicativo de linha de comando, que deve ser invocado pelo usuário a partir do arquivo `evm_loader.py`.

```
> ./evm_loader.py
```

Ao executar o programa dessa forma, estamos carregando a ontologia **GVE** na memória, porém não há nenhum cenário criado pelo usuário. Para se criar cenários, o usuário pode utilizar os seguintes argumentos:

- `-i, -individual_files`
  Passa uma lista de arquivos de onde os indivíduos de um cenário serão carregados. Exemplos:

  ```
  >./evm_loader.py -i file file2 file3
  >./evm_loader.py --individual_files file file2 file3
  ```

- `-p, -property_files`
  Passa uma lista de arquivos de onde os relacionamentos entre indivíduos de um cenário serão carregados. Exemplos:

  ```
  >./evm_loader.py -p file file2 file3
  >./evm_loader.py -property_files file file2 file3
  ```

Apesar da separação em dois arquivos, estes dois argumentos podem (e comumente devem) ser combinados para que se carregue indivíduos e relacionamentos simultaneamente, a combinação de argumentos também permite que múltiplos arquivos sejam carregados conforme os exemplos da listagem acima. Exemplos:

```
> ./evm_loader.py -i individuals_file -p property_file
> ./evm_loader.py -i a b c -p d e
```

Caso o usuário queira realizar uma navegação interativa sobre esses indivíduos, visualizando relacionamentos e navegando entre indivíduos relacionados, é possível passar a opção `-cli` na chamada do programa `evm_loader.py`. Exemplo:

```
1 > ./evm_loader.py --cli
2 > ./evm_loader.py -i individuals_file -p property_file --cli
```

O programa possui dois níveis de verbosidade, no primeiro apenas informações de inferência e erros de validação da ontologia são exibidos, no nível dois toda a ontologia carregada é exibida na tela. O primeiro nível é executado quando passamos o argumento `-v`, ou `-verbose`, e o segundo nível é executado repetindo o parâmetro. Abaixo temos exemplos de chamada do programa e seus respectivos níveis de verbosidade. Todos os níveis acima do dois possuem o mesmo comportamento do nível dois.

```
1 > ./evm_loader.py                  verbosidade=0
2 > ./evm_loader.py -v               verbosidade=1
3 > ./evm_loader.py --verbose        verbosidade=1
4 > ./evm_loader.py -v -v            verbosidade=2
5 > ./evm_loader.py -v --verbose     verbosidade=2
6 > ./evm_loader.py -vv              verbosidade=2
7 > ./evm_loader.py -vvv             verbosidade=3
```

Por fim, caso o usuário precise dessas informações de utilização do aplicativo, ele pode utilizar o parâmetro `-h`, ou `-help`. Essa opção exibe uma mensagem de ajuda contendo instruções de como utilizar o programa. Abaixo é exibido a execução do programa com esse parâmetro e a respectiva mensagem que é exibida ao usuário.

```
1 > ./evm_loader.py --help
2 usage: evm_loader [-h]
3                   [-i INDIVIDUAL_FILES [INDIVIDUAL_FILES ...]]
4                   [-p PROPERTY_FILES [PROPERTY_FILES ...]]
5                   [--cli] [-v]
6
7 This program loads the Ethical Violation Management ontology
8 definition into the memory and allows the user to initialize a set of
9 individuals and properties for this ontology from multiple files(see
10 "-i" and "-p" parameters). If no extra parameters are given, the
```

```
program will load everything, perform the ontology reasoning and
display all the relationships and entities that were both given by the
user or reasoned by the program. If the "--cli" parameter is given,
however, the program won't display those results and will start a
command line interface application instead, allowing the user to
navigate and visualize the ontology.

optional arguments:
  -h, --help            show this help message and exit
  -i INDIVIDUAL_FILES [INDIVIDUAL_FILES ...],
      --individual_files INDIVIDUAL_FILES [INDIVIDUAL_FILES ...]
                        CSV files containing ontology individuals to
                        be loaded.
  -p PROPERTY_FILES [PROPERTY_FILES ...],
      --property_files PROPERTY_FILES [PROPERTY_FILES ...]
                        CSV files containing ontology individuals
                        properties to be loaded.
  --cli                 Start the command line interface app after
                        loading the ontology
  -v, --verbose         display reasoning log messages

The evm_loader app was developed by Hugo Constantinopolos.
hugo.constantinopolos@inf.ufrgs.br
```

A.2 Comandos para navegação por linha de comando

A.2.1 Comando *help*

Comando utilizado para exibir mensagens de ajuda para utilização da navegação por linha de comando. A sintaxe do comando suporta dois tipos de utilização: a primeira, chamando unicamente o comando “help”, que traz uma mensagem geral, informando o usuário sobre o que se trata a navegação por linha de comando, e a sintaxe `help <comando>`, que exibe uma mensagem sobre utilizar um comando específico. Abaixo é exibido o resultado da execução do comando help sem nenhum argumento como argumento, e nas seções que abordam cada um dos comandos individualmente será apresentado o retorno da respectiva mensagem de ajuda.

```
Welcome to the Ethical Violation Management cli.
Type 'help' for the command reference
> help
Ethical Violation Management cli.
This program helps to visualize and analyze an ethical violation
management ontology and its individuals.

List of all available commands:
enter 'help <command>' for further help
list
select
unselect
follow
ifollow
search
show
exit
```

A.2.2 Comando *exit*

Comando utilizado para finalizar o programa, o comando “exit” não necessita de nenhum argumento, e pode ser chamado em qualquer momento.

Exemplo de utilização do comando:

```
> evm_loader.py -i individuo.csv -p propriedades.csv --cli
...

Welcome to the Ethical Violation Management cli.
Type 'help' for the command reference
> exit
INFO - Exiting
```

O usuário pode, também, utilizar o comando “help” para receber ajuda para o comando “exit”, a mensagem exibida é a seguinte:

```
> help exit
Exit the program. It can also be triggered by pressing <Ctrl+d>
```

A.2.3 Comando *list*

O comando “list” é utilizado para listar todos os indivíduos carregados no atual cenário. Sua sintaxe não permite nenhum argumento, e seu resultado apenas exibe os nomes dos indivíduos junto de um número que é seu identificador único para outras operações.

Abaixo, é exibido o resultado da execução do comando “list” utilizando o mesmo cenário criado na seção 4.8. E em seguida é exibido a mensagem de ajuda exibido ao usuário através do comando “help list”.

```
1 > list
2 [#id]    (Class) individual_name
3 [0]      (nepRobot) evm.robot
4 [1]      (evmAscriptionJustification) evm.ascription justification
5 >
```

```
1 > help list
2 Lists all individuals of the ontology and it respective ids. The list
3 command does not accept any argument.
```

A.2.4 Comando *select*

O comando “select” é utilizado para selecionar um dos indivíduos existentes na ontologia.

A operação *select* suporta dois diferentes argumentos. A primeira opção é fazer a operação a partir do número de identificação único de um indivíduo. Esse identificador é consultado através do comando *list*, conforme descrito na seção 4.9.1. Abaixo é exibido a utilização do comando *select* juntamente do identificador único de um indivíduo, assim como a de um identificador inválido.

```
1 > list
2 [#id]    (Class) individual_name
3 [0]      (nepRobot) evm.robot
4 [1]      (evmAscriptionJustification) evm.ascription justification
5 > select 99
6 individual #99 is out of range
```

```
7 > select 0
8 (evm.robot)>
```

Sobre a execução exibida acima, é importante comentar os seguintes resultados: Quando um identificador não existente(que não consta na saída do comando *list*) é passado como argumento da operação *select*, é gerado uma mensagem de erro e nenhum indivíduo é selecionado. Quando um identificador válido é passado como argumento, no entanto, o indivíduo associado ao respectivo identificador é selecionado, e isso pode ser verificado pois a linha que aguarda por comandos do usuário agora apresenta o nome do indivíduo selecionado entre parenteses.

Outra funcionalidade que a operação *select* suporta é a de selecionar um indivíduo através do seu nome. O *select* por nome do indivíduo suporta a funcionalidade de auto completar, e é *case sensitive*. Abaixo é exibido a execução do comando utilizando esse argumento:

```
1 > select invalid
2 individual <invalid> does not exist
3 > select
4 ascription justification  robot
5 > select ascription justification
6 (evm.ascription justification)>
```

Sobre a execução exibida acima, é possível notar os seguintes pontos: A operação exibe, também, uma mensagem de erro quando o usuário digita um nome de indivíduo inválido. Além disso, na linha seguinte a mensagem de erro, é exibido o comando *select* seguido de nenhum argumento, naquele momento foi pressionado a tecla <TAB>, fazendo com que o aplicativo sugerisse alternativas para completar automaticamente a linha, e o resultado é a linha seguinte com o nome dos indivíduos existentes. Na última linha, então, é exibido um indivíduo válido sendo selecionado, e o comportamento é exibir o nome do indivíduo na próxima linha, entre parenteses, antes do próximo comando do usuário.

Por fim, abaixo é exibido a mensagem de ajuda que o usuário do aplicativo vê através do comando *help*:

```
1 > help select
2 Selects an individual of the ontology.
3 usage: select <individual_id or individual_name>
```

```
4 The <individual_id> argument must be greater or equal than zero and
5 must be one of the ids listed by the 'list' command The
6 <individual_name> must exists.
```

Diferente de outros comandos apresentados até o momento, que não aceitam argumentos, essa mensagem de ajuda pode ser recebida pelo usuário caso ele digite o comando *select* sem nenhum argumento, que é uma utilização inválida do comando.

A.2.5 Comando *show*

O comando "show" é utilizado para exibir as informações de um indivíduo. Dentre as informações que são exibidas através desse comando, estão o nome do indivíduo, a classe a qual o indivíduo pertence, uma listagem dos relacionamentos diretos do indivíduo, e uma listagem com as suas propriedades inversas.

Para utilizar o comando "show" corretamente, é necessário antes selecionar um indivíduo através da operação "select" (ver seção 4.9.2). Se não houver um indivíduo selecionado, será exibido uma mensagem informando o usuário sobre essa necessidade, conforme execução abaixo:

```
1 > show
2 There is no selected individual
```

Quando há um indivíduo selecionado, no entanto, o comportamento da operação "show" é a seguinte:

```
1 > select robot A
2 (evm.robot A)> show
3
4 Name: robot A
5 Instance of: nepRobot
6
7 Properties:
8 [0] evm.robot A -> .evmis_ascribed_to ->
9                 evm.robot A responsibility ascription
10 [1] evm.robot A -> tlo.tloformulatedBy ->
11                evm.robot A ascription justification
```

```
12
13 Inverse properties:
14 [0] evm.robot A responsibility ascription -> .evmascribes ->
15                 evm.robot A
16 [1] evm.robot A ascription justification -> tlo.tloformulates ->
17                 evm.robot A
```

Abaixo é exibido a mensagem de ajuda que o usuário recebe quando utiliza o comando “help show”:

```
1 > help show
2 Show informations about the selected individual. If there is no
3 selected individual, a message informing the user will be displayed.
4 The show command will list the individual name, the class of the
5 individual and its properties with it respective ids.
```

A.2.6 Comando *unselect*

O comando “unselect”, como o nome sugere, realiza a operação de desselecionar um indivíduo. Quando o usuário realiza esta operação o indivíduo atualmente selecionado deixa esse estado, e o aplicativo seleciona o último indivíduo que esteve selecionado antes do atual, e caso não haja nenhum indivíduo selecionado, o comando não faz nada. Dessa forma, o comando “unselect” é o responsável pela funcionalidade de voltar dentro da navegação por linha de comando.

Abaixo é apresentado uma sequência de comandos “select” e “unselect” para ilustrar o funcionamento dessa navegação.

```
1 >
2 > select robot A
3 (evm.robot A)> unselect
4 > select robot A
5 (evm.robot A)> select robot B
6 (evm.robot B)> select monitor 1
7 (evm.monitor 1)> unselect
8 (evm.robot B)> unselect
9 (evm.robot A)> unselect
```

```
10 > unselect
11 >
```

A sequência de comandos acima ilustra o básico da navegação por linha de comando, especialmente a navegação através da seleção e desseleção de indivíduos no formato de pilha. Quando é realizada a primeira operação de “select”, não há nenhum indivíduo previamente selecionado, e portanto ao desselecionar o mesmo, o aplicativo volta ao estado em que não há nenhuma entidade selecionada. Na sequência, quando são realizadas sucessivas operações de “select”, são necessárias um mesmo número de operações “unselect” para retornar ao ponto inicial. Nas últimas linhas, por fim, é apresentado a execução do comando “unselect” quando não há nenhum indivíduo selecionado, e o comando não fazendo nada portanto. Abaixo é exibido a mensagem de ajuda quando solicitada através do comando “help unselect”.

```
1 > help unselect
2 Unselects an individual.
3 The unselect command will unselect the current selected individual,
4 and selected the previous selected individual, going backwards on the
5 selected individuals stack. The unselect command does not accept any
6 argument.
```

A.2.7 Comando *follow*

O comando “follow” é utilizado para realizar a navegação por entre os indivíduos que possuem um relacionamento em comum. Para isso, o comando “follow” possui a seguinte sintaxe, além de necessitar que haja um indivíduo selecionado:

```
1 follow <id_relacionamento>
```

Onde o argumento `id_relacionamento` é o identificador de uma propriedade direta do indivíduo que está atualmente selecionado, esse identificador pode ser listado através do comando “show”, conforme apresentado na seção 4.9.3. Abaixo é apresentado uma execução passo a passo do comando “follow”, auxiliado pelas funcionalidades dos comandos “show” e “unselect”:

Se utilizarmos o comando “follow”, iremos seguir o relacionamento, e iremos selecionar o indivíduo alvo da propriedade.

```
1 (evm.robot A)> follow 0
2 (evm.robot A responsibility ascription)>
```

Ou:

```
1 (evm.robot A)> follow 1
2 (evm.robot A ascription justification)>
```

Caso seja informado um identificador inválido, o usuário recebe uma mensagem informando o acontecido e a operação “follow” não acontece:

```
1 (evm.robot A)> follow 2
2 property #2 is out of range
3 (evm.robot A)>
```

Caso o usuário precise de ajuda, a seguinte mensagem é exibida através do comando “help follow”:

```
1 > help follow
2 Follow a direct property, changing the current selected individual to
3 the target of this property.
4 usage: follow <property_id>
5 The <property_id> argument must be greater or equal than zero and must
6 be one of the ids listed by the 'show' command. This command will
7 display an error if there is no selected individual.
```

A.2.8 Comando *ifollow*

O comando “ifollow” é muito semelhante ao comando “follow” (ver seção 4.9.4)porém realiza a operação de seguir uma propriedade inversa. Através do comando “ifollow”, é possível navegar a partir de um indivíduo selecionado até um indivíduo que possuí um relacionamento cujo alvo seja o indivíduo selecionado atualmente. A sintaxe do comando “ifollow” é a seguinte:

```
ifollow <id_relacionamento>
```

Onde o argumento `id_relacionamento` é o identificador de uma propriedade inversa do indivíduo que está atualmente selecionado, esse identificador pode ser listado através do comando “show”, conforme apresentado na seção 4.9.3. Como o comportamento desse comando é muito parecido com o do comando “follow”, e muito dessa navegação utilizando os comandos de seguir uma propriedade já foram apresentados na seção 4.9.4, aqui será apenas ilustrado um exemplo reduzido, exibindo o funcionamento da operação “ifollow” para navegar entre os indivíduos da ontologia, no exemplo é ilustrado toda a sequência de comandos que um usuário pode utilizar para: **a)** listar os indivíduos e selecionar um especificamente, **b)** visualizar as informações desse indivíduo, **c)** percorrer as propriedades inversas desse indivíduo e, enfim, **d)** exibir as informações desse indivíduo que foi selecionado através do comando “ifollow”.

```
> list
[#id]    (Class) individual_name
[0]      (evmNormViolation) evm.violation
[1]      (evmEthicalBehaviorMonitor) evm.monitor
> select violation
(evm.violation)> show

Name: violation
Instance of: evmNormViolation

Properties:


Inverse properties:
[0] evm.monitor -> .evmdetects -> evm.violation


(evm.violation)> ifollow 0
(evm.monitor)> show

Name: monitor
Instance of: evmEthicalBehaviorMonitor

Properties:
```

```
25 [0] evm.monitor -> .evmdetects -> evm.violation
26
27 Inverse properties:
```

Abaixo a mensagem de ajuda obtida pelo usuário através da utilização do comando “help ifollow”:

```
1 > help ifollow
2 Follow an inverse property, changing the current selected individual
3 to the source of this inverse property.
4 usage: ifollow <iproperty_id>
5 The <iproperty_id> argument must be greater or equal than zero and
6 must be one of the ids listed by the 'show' command. This command will
7 display an error if there is no selected individual.
```

A.2.9 Comando *search*

O comando “search” é responsável por realizar a busca de relacionamentos entre dois indivíduos. A sintaxe do comando é a seguinte:

```
1 search <classe>
```

Onde o argumento “classe” é o nome de uma das classes presentes na ontologia, e para que o comando funcione é necessário que haja um individuo previamente selecionado. É importante notar que as classes possíveis de serem buscadas estão disponíveis através da funcionalidade de autocompletar, ativada através da tecla <*TAB*> enquanto se digita o argumento da operação “search” Caso não haja um indivíduo selecionado, a seguinte mensagem será exibida na tela pelo aplicativo:

```
1 > search
2 There is no selected individual
```

Por fim, abaixo é exibido a mensagem de ajuda que o usuário obtém através do comando “help search”:

```
1 >help search
2 Search through the direct and inverse properties of a selected
3 individual for an individual of a given class.
4 usage: search <class_name>
5 This command will display an error message if there is no selected
6 individual.
```

É possível utilizar o comando “search” seguido de um argumento especial, o argumento “all”. Quando esse argumento é utilizado, é realizado a busca por relações de um indivíduo selecionado com indivíduos de todas as classes presentes na ontologia, agrupando os resultados por classe. Essa operação é equivalente a realizar a busca uma vez para cada classe possível de um cenário. Abaixo é ilustrado o resultado desse comando sob o cenário representado na Figura 4.4.

```
1 > select monitor
2 (evm.monitor)> search all
3
4 nepRobot
5 * evm.monitor -> .evmdetects -> evm.hit the wall -> .evmelicits ->
6 evm.robot1 responsibility ascription -> .evmascribes -> evm.robot1
7 * evm.monitor -> .evmdetects -> evm.hit another robot -> .evmelicits
8 -> evm.robot2 responsibility ascription -> .evmascribes -> evm.robot2
9
10 nepRobot
11 * evm.monitor -> .evmdetects -> evm.hit the wall -> .evmelicits ->
12 evm.robot1 responsibility ascription -> .evmascribes -> evm.robot1
13 * evm.monitor -> .evmdetects -> evm.hit another robot -> .evmelicits
14 -> evm.robot2 responsibility ascription -> .evmascribes -> evm.robot2
15
16 evmResponsibilityAscription
17 * evm.monitor -> .evmdetects -> evm.hit the wall -> .evmelicits ->
18 evm.robot1 responsibility ascription
19 * evm.monitor -> .evmdetects -> evm.hit another robot -> .evmelicits
20 -> evm.robot2 responsibility ascription
21
22 evmResponsibilityAscription
23 * evm.monitor -> .evmdetects -> evm.hit the wall -> .evmelicits ->
24 evm.robot1 responsibility ascription
25 * evm.monitor -> .evmdetects -> evm.hit another robot -> .evmelicits
```

```
26 -> evm.robot2 responsibility ascription
27
28 evmNormViolation
29 * evm.monitor -> .evmdetects -> evm.hit the wall
30 * evm.monitor -> .evmdetects -> evm.hit another robot
31
32 evmNormViolation
33 * evm.monitor -> .evmdetects -> evm.hit the wall
34 * evm.monitor -> .evmdetects -> evm.hit another robot
```

APÊNDICE B — CLASSES E RELACIONAMENTOS DA ONTOLOGIA GVE

Abaixo é apresentado a Tabela B.1 que contém todas as classes presentes que podem ser instanciadas pelo aplicativo criado nesse trabalho. Essa listagem contém não apenas as classes da ontologia **GVE**, mas contém também as classes pertencentes a outros padrões que podem manter um relacionamento com alguma das classes da ontologia **GVE**, além disso, a segunda coluna da tabela apresenta todas os relacionamentos que um indivíduo daquela classe pode possuir, e na terceira coluna qual a classe alvo desse relacionamento. Para evitar que alguns nomes como *evmLegalResponsibilityAscription* extrapolem o espaço de uma célula da tabela, esses nomes mais longos foram divididos em duas linhas e utilizam-se do caractere “-” para denotar essa quebra, apesar de seus nomes dentro da ontologia serem escritos sem esse caractere.

Nome da classe	*Propriedade*	*Classe Alvo*
evmEventCausation	evmactor evmcause evmeffect	nepAgent nepAgentAction evmNormViolation
evmResponsibilityAscription	evmascribes_distributed_- responsibility_in evmis_ascribed_to	evmResponsibilityAscription nepAgent
evmAscriptionJustification	evmcomposed_of evmjustifies	evmGroundsForAscription evmResponsibilityAscription
evmGroundsForAscription		
evmAgentAccountability		
evmNormViolationIncident	evmrecords_incidence_of	evmNormViolation
evmEthicalBehaviorMonitor	evmdetects evmobserves	evmNormViolation nepAgent
evmNormViolation	evmdocumented_in evmelicits evmis_violation_of evmrecords_incidence_of	evmNormViolationIncident evmResponsibilityAscription nepNorm evmNormViolation
evmSocioTechnology- Governance		
evmLiabilitySanction	evmis_sanction_for	evmLegalResponsibility- Ascription

Nome da classe	*Propriedade*	*Classe Alvo*
evmLegalResponsibility-Ascription	evmmay_prescribe	evmLiabilitySanction
evmAccountablePerson		
evmDistributedResponsibility		
evmEnumDataTypes		
evmEthicalResponsibility-Ascription		
nepAgent	evmascribes	evmResponsibilityAscription
	evmimposes	evmAgentAccountability
	evmmay_be_sanctioned_with	evmLiabilitySanction
	nepactivates	nepDerogation
	nepapplies	nepAgentPlan
	nepassigned_to	nepAgent
	nepcontains	nepPlanAction
	nepdeactivates	nepDerogation
	nepexecutes	nepPlanAction
	nepinitiates	tloAgentCommunication
	nepintends_to_realize	nepSituation
	nepreceives	tloAgentCommunication
	neprecognizes	nepSituation
	nepsatisfies	nepNorm
	nepselects_plans_from	nepSituationPlanRepertoire
	nepstipulates	nepNorm
	nepsubscribes_to	nepEthicalTheory
nepAgentAction	nepactivates	nepDerogation
	nepdeactivates	nepDerogation
nepGovernment	evmhas_achieved	evmSocioTechnology-Governance
	evmis_jurisdiction_of	nepAgent
nepAgentRole	evmimposes	evmAgentAccountability
	nepassigned_to	nepAgent
	nepstipulates	nepNorm
nepNorm	nepaccepted_by	nepAgent
	nepaccommodates	nepEthicalPrinciple
	nepauthorized_by	nepAgent

Nome da classe	*Propriedade*	*Classe Alvo*
	nepconflicts_with nepis_involved_in nepis_prescribed_by	nepNorm nepSituation nepEthicalTheory
nepSituation	nepfeatures_described_in nepincludes nepis_perception_of nepprecedes nepprompts nepsucceeds	nepSituationPlanRepertoire nepAgentPlan nepEnvironment nepSituation tloAgentCommunication nepSituation
nepRobot		
nepPlanAction	nepis_implemented_by	nepAgentAction
nepEthicalPrinciple		
nepDerogation	neptemporarily_suspends	nepNorm
nepAgentPlan	nepcontains nepsatisfies nepsubscribes_to	nepPlanAction nepNorm nepEthicalTheory
nepEthicalDilemma	nepcaused_by	nepNorm
nepEthicalTheory	nepconstrains_plans_for nepspecifies_norm_modality	nepSituationPlanRepertoire nepNorm
nepSituationPlanRepertoire	nepincludes	nepAgentPlan
nepTaskAssignment	nephas_as_goal	nepSituation
nepSocialCollection	nepinfluences_norm_- applicability nepprescribes_context_for	nepNorm nepNorm
nepEnvironment	nepis_perceived_by	nepAgent
nepAnswer	nepis_response_to	nepQuery
nepQuery		
nepActionRationale	neplogically_justifies	nepPlanAction
nepDilemmaMitigationPrinciple	nepranks_preferred_norm	nepEthicalDilemma
nepCommunity		
nepCompany		
nepConsequentialistNorm		
nepDeontologicalNorm		
nepDepartment		
nepEnumDataTypes		

Nome da classe	*Propriedade*	*Classe Alvo*
nepExplanation		
nepObligation		
nepOrganization		
nepPermission		
nepProhibition		
nepVirtuousNorm		
tloAgentCommunication	tlotransmits	tloInformationArtifact
tloMethod	tloisRealizedBy	tloProcess
tloAgent	tlocreates tloenacts tloformulates tlotransmits	tloActionEvent tloRole tloInformationArtifact tloInformationArtifact
tloProcess	tloaffects tlohasParticipant tlois_stage_at tlois_stage_of tlomanifests tlorealizes	tloPhysical tloevmObject tloTime tloProcess tloEvent tloMethod
tloPlan	tloselected_by	tloAgent
tloInformationArtifact	tloformulatedBy tlorenders tlotransmitted_by	tloAgent tloAbstract tloAgentCommunication
tloRole	tloenactedBy	tloAgent
tloEnumDataTypes		
tloContinuant		
tloSituation		
tloCollective	tlohasMember	tloEntity
tloPhysical	tloidentified_at tloidentified_by tlolocated_at tlopresent_at	tloTime tloProperty tloSpatioTemporalPlace tloTime
tloAbstract	tlocharacterizes	tloPhysical
tloActionEvent	tlocreated_by	tloAgent
tloEntity	tlodescribedBy tlois_member_of	tloDescription tloCollective

Nome da classe	*Propriedade*	*Classe Alvo*
	tloperceived_by	tloAgent
tloDescription	tlodescribes	tloEntity
tloEvent	tloend_time tlooccurs_in tlostart_time	tloTime tloProcess tloTime
tloTime		
tloevmObject	tloparticipates_in	tloProcess
tloProperty	tloproperty_of	tloEntity
tloSpatioTemporalPlace		
tloEnvironmentalEvent		
tloAttribute		
tloManner		
tloOccurrent		
tloInteractionProcess		
tloSchema		
tloSocialInteractionProcess		