



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Juliano Quatrin Nunes

**Grafos de Conhecimento para Integração Semântica entre Plantas Industriais e
Sistemas Multiagentes no Contexto da Web of Things**

Florianópolis
2025

Juliano Quatrin Nunes

**Grafos de Conhecimento para Integração Semântica entre Plantas Industriais e
Sistemas Multiagentes no Contexto da Web of Things**

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.

Orientador: Prof. Jomi Fred Hübner, Dr.

Florianópolis
2025

Ficha de identificação da obra

A ficha de identificação é elaborada pelo próprio autor.

Orientações em:

<http://portalbu.ufsc.br/ficha>

Juliano Quatrin Nunes

**Grafos de Conhecimento para Integração Semântica entre Plantas Industriais e
Sistemas Multiagentes no Contexto da Web of Things**

Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de
Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de
Controle e Automação

Florianópolis, 13 de Agosto de 2025.

Prof. Marcelo de Lellis Costa de Oliveira, Dr.
Coordenador do Curso

Banca Examinadora:



Documento assinado digitalmente

Jomi Fred Hubner

Data: 20/08/2025 10:00:03-0300

CPF: ***.334.769-**

Verifique as assinaturas em <https://v.ufsc.br>

Prof. Jomi Fred Hübner, Dr.
Orientador
UFSC/CTC/EAS

Prof. Jomi Fred Hübner, Dr.
Supervisor
UFSC/CTC/EAS

Prof. Gustavo Claudio Karl Couto, Dr.
Avaliador(a)
UFSC

Prof. Eduardo Camponogara, Dr.
Presidente da Banca
UFSC/CTC/EAS

AGRADECIMENTOS

Gostaria de agradecer primeiramente à minha família, meus pais Osmar Moro Nunes e Rosa Quatrin Nunes, e minha irmã, Jucielli Quatrin Nunes Hachmann, por todo o apoio e confiança em mim. Um agradecimento especial à minha namorada, Juliane Guimarães Goulart, pela paciência e companheirismo durante a jornada deste PFC.

Agradeço a todos os professores pelos conhecimentos compartilhados. Em especial, ao meu orientador, Jomi Fred Hübner, e ao meu coorientador, Iderli Souza, pela presença constante, pela ajuda no desenvolvimento e por me guiarem neste projeto.

Por fim, agradeço a todos os amigos e colegas que fiz durante a graduação, pelas boas risadas, conselhos e pelo apoio que tornaram todo o processo mais leve.

DECLARAÇÃO DE PUBLICIDADE

Florianópolis, 13 de Agosto de 2025.

Na condição de representante da Universidade Federal de Santa Catarina (UFSC), na qual o presente trabalho foi realizado, declaro não haver ressalvas quanto ao aspecto de sigilo ou propriedade intelectual sobre as informações contidas neste documento, que impeçam a sua publicação por parte da Universidade Federal de Santa Catarina (UFSC) para acesso pelo público em geral, incluindo a sua disponibilização *online* no Repositório Institucional da Biblioteca Universitária da UFSC. Além disso, declaro ciência de que o autor, na condição de estudante da UFSC, é obrigado a depositar este documento, por se tratar de um Trabalho de Conclusão de Curso, no referido Repositório Institucional, em atendimento à Resolução Normativa nº 126/2019/CUn.

Por estar de acordo com esses termos, subscrevo-me abaixo.



Documento assinado digitalmente

JULIANO QUATRIN NUNES

Data: 20/08/2025 08:37:39-0300

CPF: ***.066.589-**

Verifique as assinaturas em <https://v.ufsc.br>

Juliano Quatrin Nunes
Universidade Federal de Santa Catarina



Documento assinado digitalmente

Jomi Fred Hubner

Data: 20/08/2025 10:00:45-0300

CPF: ***.334.769-**

Verifique as assinaturas em <https://v.ufsc.br>

RESUMO

No cenário da Indústria 4.0, a necessidade de sistemas de produção flexíveis, modulares e interoperáveis impõe desafios significativos às abordagens tradicionais de automação. Este trabalho insere-se nesse contexto, partindo de um projeto anterior que explorou a integração de Sistemas Multiagentes a uma bancada industrial da FESTO por meio do padrão *Web of Things* (WoT). A motivação surge da constatação de que, mesmo com a descrição formal dos componentes fornecida pelo padrão WoT, o agente de controle ainda necessitava de conhecimento pré-programado sobre os identificadores nominais dos dispositivos para executar suas tarefas, limitando a adaptabilidade do sistema a mudanças de hardware. Portanto, o objetivo deste trabalho é propor e validar uma arquitetura que supera essa limitação, permitindo que o agente descubra e interaja com os dispositivos com base em suas capacidades funcionais, em vez de seus nomes estáticos. O método central para alcançar tal desacoplamento foi a modelagem da bancada didática Separating como um Grafo de Conhecimento. Este processo envolveu a representação formal da planta, seus subsistemas e componentes individuais em *Resource Description Framework* (RDF), fundamentando-se em ontologias padrão da indústria, como *Sensor Object Schema* (SOSA) para a estrutura física e *Smart Application Reference Ontology* (SAREF) para as capacidades funcionais, garantindo assim a interoperabilidade do modelo. A implementação da solução utilizou a plataforma Apache Jena para a construção e persistência do grafo, o qual foi exposto via *Hypertext Transfer Protocol* (HTTP) por um servidor Linked Data para ser consumido por um agente inteligente. Este agente, desenvolvido em Jason e integrado pelo framework Hypermedea, utiliza o grafo como sua única fonte de conhecimento sobre a infraestrutura. Como resultado principal, a validação da abordagem demonstrou que o agente é capaz de se adaptar a modificações de hardware sem modificação em seu código. O impacto desta abordagem, fundamentada na Web Semântica, reside na efetiva substituição da lógica de seleção de dispositivos, que transita de uma referência estática para uma descoberta dinâmica, reduzindo o custo e o tempo de reconfiguração do sistema. Embora a lógica sequencial do processo produtivo permaneça definida nos planos do agente, o trabalho valida um avanço concreto em direção a sistemas de automação mais resilientes e flexíveis, sendo especialmente aplicáveis a contextos de automação dinâmicos, onde a frequência de alteração da infraestrutura de sensores e atuadores é muito alta.

Palavras-chave: Sistemas Multiagentes. Web Semântica. Grafos de Conhecimento.

ABSTRACT

In the Industry 4.0 scenario, the need for flexible, modular, and interoperable production systems imposes significant challenges on traditional automation approaches. This work is set within this context, stemming from a previous project that explored the integration of Multi-Agent Systems with a FESTO industrial bench through the WoT standard. The motivation arises from the observation that, even with the formal component descriptions provided by the WoT standard, the control agent still required pre-programmed knowledge of the nominal identifiers of the devices to execute its tasks, limiting the system's adaptability to hardware changes. Therefore, the objective of this work is to propose and validate an architecture that overcomes this limitation, allowing the agent to discover and interact with devices based on their functional capabilities instead of their static names. The central method to achieve such decoupling was the modeling of the Separating didactic bench as a Knowledge Graph. This process involved the formal representation of the plant, its subsystems, and individual components in RDF, grounded in standard industry ontologies such as SOSA for the physical structure and SAREF for the functional capabilities, thus ensuring the model's interoperability. The solution's implementation utilized the Apache Jena platform for the construction and persistence of the graph, which was exposed via HTTP by a Linked Data server to be consumed by an intelligent agent. This agent, developed in Jason and integrated with the Hypermedea framework, uses the graph as its sole source of knowledge about the infrastructure. As a main result, the validation of the approach demonstrated that the agent is capable of adapting to hardware modifications without any changes to its code. The impact of this approach, based on the Semantic Web, lies in the effective replacement of the device selection logic, which transitions from a static reference to a dynamic discovery, reducing the cost and time of system reconfiguration. Although the sequential logic of the production process remains defined in the agent's plans, the work validates a concrete advancement towards more resilient and flexible automation systems, being especially applicable to dynamic automation contexts where the frequency of changes in the sensor and actuator infrastructure is very high.

Keywords: Multi-Agent Systems. Semantic Web. Knowledge Graphs.

LISTA DE FIGURAS

Figura 1 – Esquemático da bancada <i>Separating</i> com seus componentes numerados.	26
Figura 2 – Arquitetura da Solução Proposta	37
Figura 3 – Arquitetura da Camada de Conhecimento	44
Figura 4 – Relação hierárquica entre a linha de produção MPSFesto e a bancada <i>Separating</i>	46
Figura 5 – Grafo de conhecimento do subsistema da esteira principal.	48
Figura 6 – Grafo de conhecimento do subsistema de verificação de orientação.	49
Figura 7 – Grafo de conhecimento do subsistema de descarte.	50
Figura 8 – Grafo de conhecimento resumido do sensor de orientação.	52
Figura 9 – Estrutura de pacotes do projeto Java.	53
Figura 10 – Diagrama de classes da arquitetura de construção do grafo.	57
Figura 11 – Diagrama de sequência para uma requisição ao Servidor <i>Linked Data</i>	60
Figura 12 – Sequência de eventos do ciclo de passagem na planta física.	69
Figura 13 – Sequência de eventos do ciclo de descarte na planta física.	70

LISTA DE TABELAS

Tabela 2 – Análise comparativa de abordagens para a camada de conhecimento.	41
Tabela 3 – Sequência de eventos principais da execução com o grafo original.	68

LISTA DE ABREVIATURAS E SIGLAS

BDI	<i>Belief, Desire, Intention</i>
CLP	Controlador Lógico Programável
HTTP	<i>Hypertext Transfer Protocol</i>
IoT	<i>Internet of Things</i>
JSON	Notação de Objetos JavaScript
JSON-LD	JSON para Dados Vinculados
LAI	Laboratório de Automação e Informática Industrial
OPC UA	<i>Open Platform Communications Unified Architecture</i>
RDF	<i>Resource Description Framework</i>
SAREF	<i>Smart Application Reference Ontology</i>
SMA	Sistema Multiagentes
SOSA	<i>Sensor Object Schema</i>
SSN	<i>Sensor Service Network</i>
TD	<i>Thing Description</i>
Turtle	<i>Terse RDF Triple Language</i>
UFSC	Universidade Federal de Santa Catarina
URI	Identificador Uniforme de Recurso
W3C	<i>World Wide Web Consortium</i>
WoT	<i>Web of Things</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	OBJETIVOS	15
1.1.1	Objetivo Geral	15
1.1.2	Objetivos Específicos	15
1.2	ESTRUTURA DO DOCUMENTO	16
2	FUNDAMENTAÇÃO TEÓRICA	17
2.1	<i>WEB OF THINGS</i> (WOT)	17
2.1.1	Thing Descriptions (TDs)	17
2.2	TECNOLOGIAS DA WEB SEMÂNTICA	18
2.2.1	Grafos de Conhecimento	18
2.2.2	Resource Description Framework (RDF)	19
2.2.3	Representações RDF	20
2.2.4	Frameworks para Manipulação de Grafos RDF: Apache Jena	21
2.3	ONTOLOGIAS PARA AUTOMAÇÃO INDUSTRIAL	22
2.4	SISTEMAS MULTIAGENTES EM AMBIENTES INDUSTRIAIS	22
2.4.1	Arquitetura BDI e a Plataforma JaCaMo	23
2.4.2	Automação Industrial Baseada em Agentes	23
2.4.3	Apresentação do <i>framework</i> Hypermedea	24
3	DESCRIÇÃO DO PROBLEMA E REQUISITOS TÉCNICOS	25
3.1	AMBIENTE DO LABORATÓRIO LAI	25
3.1.1	Descrição da Linha de Montagem	25
3.1.2	A Bancada Separating como Objeto de Estudo	25
3.2	AMBIENTE DE SIMULAÇÃO PARA DESENVOLVIMENTO	26
3.3	PROJETO “AGENTS IN CHARGE”	27
3.3.1	Arquitetura do Sistema	27
3.3.2	Middleware Baseado em Node-RED	28
3.3.3	Implementação das <i>Thing Description</i> (TD)	28
3.4	LIMITAÇÕES E DESAFIOS ATUAIS	29
3.4.1	Alto Acoplamento Entre Agentes e Dispositivos	29
3.4.2	Falta de Compreensão Semântica	30
3.4.3	Escalabilidade e Reusabilidade Limitadas	32
3.5	REQUISITOS TÉCNICOS	33
3.5.1	Requisitos Funcionais	33
3.5.2	Requisitos Não-Funcionais	34
3.5.3	Restrições de Projeto	34
4	SOLUÇÃO PROPOSTA E METODOLOGIA UTILIZADA	36
4.1	VISÃO GERAL DA SOLUÇÃO PROPOSTA	36

4.2	METODOLOGIA DE PROJETO	37
4.2.1	Princípios de Design para a Modelagem Semântica	38
4.2.2	Definição da Interface Semântica via Ontologias Padrão	38
4.2.3	Procedimentos de Verificação e Validação	39
4.3	ARQUITETURA DA SOLUÇÃO E TECNOLOGIAS PROPOSTAS . .	40
4.3.1	Análise de Abordagens para a Implementação do Grafo	40
4.3.1.1	Arquivo RDF Estático	40
4.3.1.2	JSON-LD Estático (TD Expandido)	41
4.3.1.3	Banco Relacional e Conversão para RDF	42
4.3.1.4	Apache Jena com TDB2	42
4.3.2	Arquitetura da Camada de Conhecimento	43
5	DESENVOLVIMENTO E ANÁLISE DOS RESULTADOS OBTIDOS .	45
5.1	MODELAGEM SEMÂNTICA DA BANCADA SEPARATING	45
5.1.1	Diagrama do Grafo de Conhecimento da Bancada	45
5.1.1.1	Modelagem do Subsistema da Esteira Principal	47
5.1.1.2	Modelagem do Subsistema de Verificação de Orientação	48
5.1.1.3	Modelagem do Subsistema de Descarte	49
5.1.2	Integração do TD com o Grafo de Conhecimento	49
5.1.3	Princípios de Design e Estrutura da URI	51
5.2	IMPLEMENTAÇÃO DA CAMADA DE CONHECIMENTO COM APA- CHE JENA	52
5.2.1	Estrutura do Projeto Java	53
5.2.2	Importação de Ontologias	53
5.2.3	Construção Programática do Grafo	56
5.2.3.1	A Classe <i>Vocabulary</i>	56
5.2.3.2	Arquitetura de Classes para a Construção do Grafo	56
5.2.3.3	Exemplos de Implementação de Componentes	57
5.2.3.4	O Orquestrador TDBBuilder	59
5.2.4	Serialização e Exposição do Grafo Gerado	60
5.3	ORQUESTRAÇÃO DE SERVIÇOS COM DOCKER	61
5.3.1	Arquitetura de Contêineres e o Docker Compose	61
5.3.2	Replicação e Perfis de Execução	62
5.4	IMPLEMENTAÇÃO E OPERAÇÃO DO AGENTE DE CONTROLE . .	62
5.4.1	Estrutura do Agente BDI em JaCaMo/Hypermedea	63
5.4.2	O Ciclo de Descoberta Semântica em Ação	63
5.5	VERIFICAÇÃO DE REQUISITOS E ANÁLISE DE RESULTADOS . .	65
5.5.1	Verificação do Atendimento aos Requisitos	65
5.5.2	Validação do Desacoplamento Semântico	66
5.5.2.1	Execução com o Grafo de Conhecimento Original	67

5.5.2.2	Execução com o Grafo de Conhecimento Modificado	68
5.5.3	Validação em Ambiente Físico	68
5.5.4	Análise Crítica e Limitações da Solução	70
6	CONCLUSÃO	72
6.1	TRABALHOS FUTUROS	72
	REFERÊNCIAS	74
	APÊNDICE A – CÓDIGO FONTE DO SERVIDOR <i>LINKED DATA</i> .	78

1 INTRODUÇÃO

A Indústria 4.0 é marcada pela crescente digitalização dos processos de manufatura e pela integração entre sistemas ciber-físicos (Kagermann; Wahlster; Helbig, 2013). Nesse novo paradigma, a interoperabilidade — a capacidade de diferentes sistemas operarem de forma coordenada — tornou-se um dos pilares fundamentais para atingir maior eficiência, flexibilidade e adaptabilidade nos ambientes produtivos (Hermann; Pentek; Otto, 2016).

Nesse contexto, a utilização de agentes inteligentes tem se destacado como uma abordagem promissora para a automação de processos industriais (Leitão, 2009; Leitão et al., 2016). Esses agentes, dotados de capacidades de percepção, raciocínio e ação, são capazes de operar de forma autônoma em ambientes dinâmicos e distribuídos, favorecendo a tomada de decisão descentralizada e a adaptação a mudanças operacionais.

Este trabalho foi inspirado nas diretrizes discutidas durante a *AI4Industry Summer School*, uma iniciativa voltada à aplicação de inteligência artificial em sistemas industriais ciber-físicos, promovendo o uso de tecnologias como agentes autônomos, grafos de conhecimento e ontologias industriais para fortalecer a interoperabilidade semântica (Nardin, 2024). Os conceitos apresentados nesse evento serviram como referência para a proposta aqui desenvolvida.

A pesquisa foi realizada no Laboratório de Automação e Informática Industrial (LAI) da Universidade Federal de Santa Catarina (UFSC), um laboratório que oferece infraestrutura para disciplinas e atividades relacionadas à automação industrial. Entre os recursos disponíveis está uma bancada didática da linha de produção modular (MPS) da FESTO, frequentemente utilizada no ensino de programação de Controlador Lógico Programáveis (CLPs) e em projetos de pesquisa. Essa bancada permite simular um processo de manufatura em pequena escala, servindo como plataforma para experimentação e validação de soluções alinhadas aos princípios da Indústria 4.0.

O ponto de partida para este trabalho foi a análise do projeto “Agents in Charge”, desenvolvido anteriormente na disciplina de Projeto Integrador. O projeto estabeleceu uma base de comunicação entre agentes inteligentes e a bancada *Separating*, utilizando o padrão WoT. No entanto, uma limitação importante foi identificada: o alto acoplamento semântico entre o agente de controle e os dispositivos da planta. Apesar da interface padronizada, o agente ainda precisava conhecer previamente os identificadores nominais dos dispositivos para operar corretamente, o que o tornava sensível a mudanças no hardware ou na implementação do servidor WoT — um problema relevante no contexto industrial, onde reconfigurações rápidas e de baixo custo são essenciais para manter a competitividade.

Para endereçar esta limitação, o presente trabalho apresenta como solução o

desenvolvimento de uma arquitetura que complementa a infraestrutura WoT existente com uma camada de conhecimento semântico. A proposta central é a utilização de um Grafo de Conhecimento, modelado em RDF, que atua como uma fonte única de informação sobre a planta. A metodologia para a sua construção envolveu a seleção e integração de ontologias padrão da indústria, como SOSA, SAREF e a ontologia do projeto *AI4Industry* (Nardin, 2024), para descrever a hierarquia, as capacidades funcionais e as interfaces de interação dos dispositivos.

A implementação consistiu no desenvolvimento de uma aplicação Java com o framework Apache Jena para a construção programática e persistência do grafo em um banco de dados de triplas. Adicionalmente, foi implementado um servidor *Linked Data* para expor o grafo via HTTP. Para validar a abordagem, foi utilizado um agente de software, que consome o grafo para descobrir e operar os dispositivos. O principal resultado obtido, validado em um ambiente simulado, foi a comprovação de que o agente consegue se adaptar a mudanças no hardware (como a substituição de um sensor) apenas consultando o grafo atualizado, sem nenhuma alteração em seu código-fonte.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Desenvolver uma arquitetura para exposição semântica de dispositivos em bancadas industriais do LAI, integrando descrições no padrão WoT a um Grafo de Conhecimento em RDF, com o objetivo de viabilizar a interpretação e a interação autônoma por agentes inteligentes, reduzindo o acoplamento entre a planta e o agente de controle.

1.1.2 Objetivos Específicos

- Selecionar e integrar ontologias industriais padrão para descrever os componentes da planta.
- Modelar um Grafo de Conhecimento em RDF que represente a hierarquia, as capacidades funcionais e as interfaces de interação da bancada *Separating*.
- Desenvolver uma arquitetura de software para a construção, persistência e exposição do Grafo de Conhecimento.
- Validar a solução por meio de uma prova de conceito com um agente *Belief, Desire, Intention* (BDI), capaz de consumir o grafo para descobrir e operar dispositivos de forma autônoma.

1.2 ESTRUTURA DO DOCUMENTO

Este documento está organizado da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica sobre os principais conceitos e técnicas necessários para o entendimento do problema abordado e da solução proposta. O Capítulo 3 descreve em detalhes o problema tratado e os requisitos técnicos a serem atendidos. O Capítulo 4 apresenta a solução proposta e a metodologia empregada. O Capítulo 5 relata o desenvolvimento realizado e analisa os resultados obtidos. Por fim, o Capítulo 6 apresenta as conclusões do trabalho e propõe direções para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os conceitos teóricos necessários para a compreensão do trabalho desenvolvido. São abordados temas relacionados à WoT, tecnologias da web semântica, ontologias industriais e sistemas multiagentes aplicados à automação industrial.

2.1 WEB OF THINGS (WOT)

A WoT representa uma camada de abstração sobre a *Internet of Things* (IoT) que padroniza como dispositivos físicos são expostos e acessados através da Web. Desenvolvida pelo *World Wide Web Consortium* (W3C), a WoT fornece mecanismos para integração e interoperabilidade de dispositivos IoT heterogêneos através de tecnologias web estabelecidas, tais como HTTP, Notação de Objetos JavaScript (JSON) e Identificador Uniforme de Recurso (URI) (W3C, 2020b).

O diferencial dessa abordagem torna-se evidente ao contrastá-la com a IoT tradicional, que frequentemente depende de protocolos proprietários e cria barreiras de integração. Em oposição, a WoT eleva o foco para a interoperabilidade semântica, o que é especialmente crítico em ambientes industriais com sistemas de múltiplos fabricantes. Essa padronização reduz significativamente o esforço de integração e permite que desenvolvedores apliquem competências web existentes no domínio da automação industrial (Korkan; Kaebisch; Steinhorst, 2020; Zeng; Guo; Cheng, 2011).

2.1.1 Thing Descriptions (TDs)

Uma TD é um documento em JSON para Dados Vinculados (JSON-LD) que descreve uma *Thing*, que por sua vez é uma abstração de uma entidade física ou virtual que participa da WoT. Essa descrição inclui metadados como nome, descrição, propriedades, ações e eventos. Em essência, ela funciona como um guia de como uma *Thing* pode ser acessada e interagida através da Web. Essa descrição é fundamental para a interoperabilidade e a integração de dispositivos WoT em ambientes industriais heterogêneos (W3C, 2020c).

O modelo de uma TD estrutura-se em torno de três tipos de interações, conhecidas como *affordances*: *Properties* (estados observáveis), *Actions* (invocações que alteram estados) e *Events* (notificações assíncronas). Essa estrutura é fundamentada em uma ontologia formal desenvolvida pelo W3C, que estabelece um vocabulário padronizado para garantir uma representação consistente e interoperável de dispositivos heterogêneos (W3C, 2020c).

A ontologia define um vocabulário com classes e propriedades centrais, incluindo:

- **Classes:** `td:Thing` e `td:InteractionAffordance`, com suas especializações `td:PropertyAffordance`, `td:ActionAffordance` e `td:EventAffordance`.
- **Propriedades:** Termos para metadados (`td:title`, `td:description`), segurança (`td:securityDefinitions`) e, crucialmente, para os protocolos de comunicação através da propriedade `td:hasForm`, que detalha os endpoints e operações (ex: `http:methodName`) para cada interação.

A representação em JSON-LD combina a simplicidade do JSON com a expressividade semântica do RDF, permitindo tanto processamento por aplicações convencionais quanto integração com tecnologias da web semântica (W3C, 2020a). Através do mecanismo de contexto do JSON-LD, as TDs podem ser estendidas com vocabulários de domínio específico, possibilitando integração com ontologias industriais estabelecidas como *Sensor Service Network* (SSN) e SOSA para redes de sensores ou SAREF para automação. Conforme demonstrado por Charpenay e Käbis (2020), o alinhamento da ontologia TD com esses outros vocabulários é essencial para garantir a interoperabilidade semântica e posicionar as TDs como uma ponte fundamental entre o mundo físico

2.2 TECNOLOGIAS DA WEB SEMÂNTICA

A Web Semântica representa uma extensão da Web tradicional onde a informação é estruturada com significado bem definido, permitindo que máquinas e humanos trabalhem em cooperação (Berners-Lee, Tim; Hendler; Lassila, 2001). Proposta por Tim Berners-Lee, Hendler e Lassila (2001), visa transformar a web de documentos em uma web de dados interligados, onde o significado é explicitamente representado através de ontologias e vocabulários padronizados.

No contexto industrial, a Web Semântica endereça desafios críticos de heterogeneidade de dados em sistemas cyber-físicos. Como demonstrado por Jirkovsky, Obitko e Marik (2017), a interoperabilidade semântica é essencial para a integração efetiva de sistemas IoT industriais, reduzindo custos de integração e permitindo colaboração entre diferentes stakeholders da cadeia de valor.

2.2.1 Grafos de Conhecimento

Grafos de conhecimento representam informações complexas através de entidades (nós) conectadas por relações semânticas (arestas), formando redes navegáveis de dados interconectados. Conforme Ji et al. (2021), estas estruturas caracterizam-se por sua natureza heterogênea, extensibilidade incremental e capacidade de capturar incerteza e proveniência através de anotações.

As vantagens dos grafos na representação de relações complexas incluem a modelagem natural de relações n-árias e hierárquicas, facilidade de integração de fontes heterogêneas e possibilidade de inferência através de raciocínio automatizado. A natureza visual dos grafos também facilita a compreensão humana enquanto mantém o processamento computacional eficiente (Ji et al., 2021).

Em contextos industriais, grafos de conhecimento são aplicados na manutenção preditiva, integrando dados de sensores, históricos de manutenção e especificações técnicas para identificar padrões e prever falhas. Na otimização de cadeias de suprimentos, representam relações complexas entre fornecedores, produtos e processos, e na gestão de qualidade, conectam parâmetros de processo com resultados de inspeção. Buchgeher et al. (2019) demonstram como essas aplicações reduzem o tempo de inatividade e melhoram a eficiência operacional em ambientes de manufatura.

Exemplos relevantes incluem grafos de conhecimento industriais da Siemens para manufatura inteligente, aplicações em gestão de qualidade que conectam parâmetros de processo com resultados de inspeção, e sistemas de rastreabilidade que mapeiam toda a cadeia produtiva. Estas implementações demonstram o valor prático da tecnologia em cenários reais de produção (Buchgeher et al., 2019).

2.2.2 Resource Description Framework (RDF)

O RDF constitui o modelo de dados fundamental da Web Semântica, fornecendo uma estrutura flexível e extensível para representar informações sobre recursos de forma processável por máquinas. Desenvolvido pelo W3C, o RDF baseia-se em um modelo de grafos direcionados que permite expressar relações entre recursos através de uma sintaxe uniforme (W3C, 2014a).

O modelo de dados RDF fundamenta-se no conceito de triplas, onde cada declaração consiste em três componentes: sujeito (recurso sobre o qual se faz uma afirmação), predicado (propriedade ou relação) e objeto (valor ou outro recurso relacionado). Esta estrutura simples, mas poderosa, permite a construção incremental de grafos de conhecimento complexos (Klyne; Carroll, 2004). O Código 2.1 apresenta uma declaração RDF que descreve um sensor de temperatura.

```
1 <http://factory.example/sensor/temp01>  
2   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  
3   <http://example.org/TemperatureSensor> .  
4 <http://factory.example/sensor/temp01>  
5   <http://example.org/hasPrecision> "0.1"^^xsd:float .
```

Código 2.1 – Descrição de um sensor em RDF

Ontologias RDF padronizadas são essenciais para interoperabilidade semântica. No contexto industrial, destacam-se a SSN para redes de sensores (Compton et al., 2012), SAREF para aplicações inteligentes, além de vocabulários fundamentais como

o RDF Schema. Essas ontologias e vocabulários fornecem termos reutilizáveis que promovem compreensão compartilhada entre sistemas heterogêneos.

2.2.3 Representações RDF

A serialização de dados RDF em formatos textuais viabiliza o intercâmbio, armazenamento e processamento de informações semânticas. Diferentes formatos atendem a necessidades específicas, desde legibilidade humana até integração com tecnologias web existentes (W3C, 2014b).

O *Terse RDF Triple Language* (Turtle) foi desenvolvido como um subconjunto simplificado da *Notation3* (N3), uma linguagem mais expressiva para a Web Semântica que também permite a definição de regras. herdando características do N3, como o uso de prefixos para abreviar URIs e atalhos sintáticos para padrões comuns, o Turtle foca exclusivamente em oferecer uma sintaxe concisa e legível para expressar triplas RDF (Beckett; Berners-Lee, Tim, 2011). O Código 2.2 apresenta uma descrição de um sensor de temperatura em Turtle.

```
1 @prefix td: <https://www.w3.org/2019/wot/td#> .
2 @prefix sosa: <http://www.w3.org/ns/sosa/> .
3
4 :sensor01 a td:Thing, sosa:Sensor ;
5     td:title "Sensor de Temperatura Industrial"@pt ;
6     sosa:observes :temperatura_sala_101 ;
7     td:hasPropertyAffordance [
8         td:name "temperatura" ;
9         td:hasForm [ td:href "mqtt://broker.local/temp01" ]
10    ] .
```

Código 2.2 – Descrição de um sensor em Turtle

JSON-LD permite anotar documentos JSON com semântica através do mecanismo `@context`, mantendo compatibilidade com aplicações JSON existentes. Esta característica é crucial para adoção em APIs REST e aplicações web modernas (Sporny et al., 2020). O Código 2.3 apresenta uma descrição de um sensor de temperatura em JSON-LD.

```
1 {
2   "@context": [ "https://www.w3.org/2019/wot/td/v1", {
3     "sosa": "http://www.w3.org/ns/sosa/"
4   } ],
5   "@type": [ "Thing", "sosa:Sensor" ],
6   "title": "Sensor de Temperatura Industrial",
7   "sosa:observes": { "@id": "temperatura_sala_101" },
8   "properties": {
9     "temperatura": {
10       "forms": [ { "href": "mqtt://broker.local/temp01" } ]
11     }
12   }
13 }
```

```
11     }  
12 }  
13 }
```

Código 2.3 – Exemplo de JSON-LD

A comparação entre representações revela trade-offs importantes. Turtle oferece concisão e clareza para usuários familiarizados com RDF, sendo ideal para ontologias e configurações. JSON-LD, embora mais verboso, integra-se naturalmente com ferramentas de desenvolvimento web, sendo a escolha predominante para TDs na WoT. Como observa Champin (2014), ambos os formatos são completamente interoperáveis, permitindo conversão sem perda de informação semântica.

2.2.4 Frameworks para Manipulação de Grafos RDF: Apache Jena

Para a manipulação programática de grafos RDF, diversos frameworks foram desenvolvidos. No ecossistema Java, um dos mais robustos e consolidados é o Apache Jena, um framework de código aberto que oferece um conjunto completo de ferramentas para a construção de aplicações para a Web Semântica (The Apache Software Foundation, 2025).

O principal componente do Jena é sua RDF API, que permite a criação e manipulação de grafos em memória. A interface `Model` representa o grafo como um contêiner de triplas. Cada tripla, ou `Statement`, é composta por um `Resource` (o sujeito), uma `Property` (o predicado) e um `RDFNode` (o objeto, que pode ser outro `Resource` ou um `Literal`). A construção de uma tripla é realizada de forma intuitiva, como ilustra o Código 2.4, que mostra um exemplo genérico.

```
1 // Criar um modelo RDF em memória  
2 Model model = ModelFactory.createDefaultModel();  
3 String ns = "http://example.org/ns#"; // Namespace de exemplo  
4 // Criar os recursos e a propriedade  
5 Resource book = model.createResource(ns + "book1");  
6 Property title = model.createProperty(ns + "title");  
7 Literal bookTitle = model.createLiteral("Introduction to RDF", "en");  
8 // Adicionar a tripla (Statement) ao modelo  
9 model.add(book, title, bookTitle);
```

Código 2.4 – Exemplo genérico de construção de uma tripla com a RDF API do Jena.

Além da manipulação em memória, o ecossistema Jena oferece soluções para a persistência de grafos. O componente TDB2 é um motor de armazenamento de triplas de alta performance, projetado para atuar como um banco de dados embarcado. Ele permite que grafos de grande volume sejam armazenados eficientemente em disco e acessados pela aplicação, formando a base para a construção de um Triple Store.

2.3 ONTOLOGIAS PARA AUTOMAÇÃO INDUSTRIAL

Para que sistemas computacionais possam trocar informações sem ambiguidades, é necessário um modelo que defina formalmente os conceitos de um domínio. Esse modelo é chamado de ontologia. Ontologias representam especificações formais e explícitas de conceitualizações compartilhadas, fornecendo vocabulários estruturados que definem entidades, propriedades e relações em um domínio específico. No contexto de sistemas industriais, ontologias são fundamentais para garantir interoperabilidade semântica entre dispositivos heterogêneos, permitindo que diferentes sistemas compartilhem e interpretem informações de forma consistente (Jirkovsky; Obitko; Marik, 2017).

Os princípios de modelagem ontológica para automação industrial incluem modularidade (separação em componentes reutilizáveis), extensibilidade (capacidade de adicionar novos conceitos) e alinhamento com padrões estabelecidos. A abordagem de reuso e extensão de ontologias existentes é essencial para evitar redundâncias e promover interoperabilidade. Como demonstrado por Compton et al. (2012) com a SSN, ontologias bem projetadas servem como pontos de integração para vocabulários específicos de domínio.

O panorama de ontologias para automação industrial inclui vocabulários estabelecidos como SSN/SOSA para redes de sensores (Compton et al., 2012), SAREF e suas extensões para domínios específicos, e ontologias de unidades de medida como QUDT e OM. Ontologias emergentes como a Building Topology Ontology (BOT) e vocabulários específicos de manufatura complementam este ecossistema.

Os critérios para seleção de ontologias em projetos industriais incluem: maturidade e adoção pela comunidade, alinhamento com a necessidade do projeto e facilidade de integração com sistemas existentes. Deve-se levar em consideração também que, em contextos industriais específicos, pode ser necessário expandir ontologias existentes, criando especializações que melhor representam o ambiente desejado.

2.4 SISTEMAS MULTIAGENTES EM AMBIENTES INDUSTRIAIS

Sistema Multiagentes (SMA) representam uma abordagem computacional onde múltiplos agentes autônomos interagem para resolver problemas complexos através de cooperação, coordenação e negociação, tendo emergido no contexto industrial como um paradigma para endereçar desafios de flexibilidade, adaptabilidade e descentralização em sistemas de manufatura modernos (Bordini; Hübner; Wooldridge, 2007).

A aplicação de SMAs na indústria evoluiu desde sistemas de controle distribuído nos anos 1990 até arquiteturas sofisticadas para Indústria 4.0. Esta evolução foi impulsionada pela necessidade de sistemas de produção mais ágeis, capazes de auto-organização e reconfiguração dinâmica em resposta a mudanças de demanda e

falhas de equipamentos (Leitão et al., 2016; Leitão, 2009).

2.4.1 Arquitetura BDI e a Plataforma JaCaMo

A arquitetura BDI fornece um modelo cognitivo para agentes racionais baseado em estados mentais que simulam o raciocínio prático humano. Nesse modelo, as Beliefs (crenças) representam o conhecimento que o agente possui sobre si mesmo e sobre o ambiente. Os Desires (desejos) correspondem aos objetivos ou estados que o agente gostaria de alcançar. Já as Intentions (intenções) são os desejos que o agente se compromete a perseguir ativamente, representando os objetivos selecionados para os quais ele irá formular e executar um plano. Este modelo permite o desenvolvimento de agentes com um equilíbrio entre comportamento deliberativo (guiado por intenções) e reativo (em resposta a mudanças nas crenças) (Rao; Georgeff, 1995).

Uma das principais implementações dessa arquitetura é a plataforma JaCaMo, que integra três tecnologias complementares para o desenvolvimento de sistemas multiagentes: Jason, para a programação do raciocínio autônomo dos agentes; CArtAgO, para a programação do ambiente de execução na forma de artefatos; e Moise, para a especificação da dimensão organizacional do sistema. O foco deste trabalho recai sobre Jason, que implementa a linguagem AgentSpeak, uma extensão operacional da arquitetura BDI. Conforme demonstrado por Bordini, Hübner e Wooldridge (2007), Jason fornece uma sintaxe de alto nível para a programação de agentes, incluindo construções para crenças (átomos Prolog), objetivos (precedidos por ! ou ?) e planos (regras ativadas por eventos).

O ciclo de raciocínio de um agente BDI em Jason opera da seguinte forma: (1) o agente percebe o ambiente e atualiza suas crenças; (2) mudanças nas crenças geram novos eventos; (3) o agente seleciona planos aplicáveis para lidar com os eventos e seus desejos; (4) uma vez que um plano é selecionado para um objetivo, ele se torna uma intenção; (5) o agente executa as ações contidas na intenção. Este ciclo permite que os agentes respondam a mudanças dinâmicas no ambiente, mantendo ao mesmo tempo o compromisso com seus objetivos de longo prazo, uma característica essencial para aplicações em ambientes industriais (Bordini; Hübner; Wooldridge, 2007).

2.4.2 Automação Industrial Baseada em Agentes

Paradigmas de controle baseado em agentes para automação industrial incluem arquiteturas holônicas, sistemas de manufatura bioinspirados e organizações de agentes normativos. Estes paradigmas compartilham características de autonomia local, comunicação peer-to-peer e emergência de comportamento global através de interações locais (Leitão, 2009).

Diferentemente de abordagens tradicionais hierárquicas e centralizadas, sistemas baseados em agentes distribuem inteligência e tomada de decisão. Enquanto

sistemas SCADA tradicionais dependem de controladores mestres, SMAs permitem que cada recurso de produção tome decisões locais, aumentando robustez e escalabilidade. Esta descentralização facilita plug-and-produce de novos equipamentos (Monostori; Kádár; Bauernhansl et al., 2016).

A integração com dispositivos físicos ocorre através de agentes wrapper que encapsulam funcionalidades de CLPs, sensores e atuadores, expondo-as através de interfaces padronizadas. Protocolos como OPC-UA e, mais recentemente, WoT, facilitam esta integração fornecendo abstrações uniformes sobre heterogeneidade de hardware (Leitão et al., 2016).

Entretanto, desafios de interoperabilidade semântica persistem devido a diferentes modelos de informação entre fabricantes e domínios. A combinação de agentes com ontologias e tecnologias de web semântica emerge como solução promissora, permitindo que agentes compartilhem entendimento comum sobre capacidades e estados de dispositivos (Monostori; Kádár; Bauernhansl et al., 2016).

2.4.3 Apresentação do *framework* Hypermedea

Hypermedea representa uma extensão do framework JaCaMo para desenvolvimento de agentes capazes de atuar em ambientes Web e WoT. Conforme apresentado por Charpenay et al. (2022), o framework permite que agentes BDI programados em Jason naveguem e raciocinem sobre dados semânticos expostos através de princípios Linked Data.

A integração com web semântica ocorre através de artefatos CArtAgO especializados. O Linked Data artifact gerencia navegação por recursos RDF, mantendo grafos nomeados em memória e expondo triplas como propriedades observáveis (rdf/3). O Ontology artifact realiza inferência OWL incremental, materializando asserções derivadas como predicados acessíveis aos agentes. Esta arquitetura permite que agentes Jason percebam e raciocinem sobre conhecimento semântico sem modificação da linguagem base (Charpenay et al., 2022).

A obtenção de conhecimento RDF segue o paradigma de navegação Linked Data: agentes decidem autonomamente quais URIs visitar através da operação visit/1, recebem representações RDF como crenças, e podem seguir links para descobrir novos recursos. O Thing artifact complementa esta capacidade fornecendo operações para interagir com TDs, permitindo que agentes leiam propriedades, invoquem ações e subscrevam a eventos de dispositivos WoT. Esta combinação de raciocínio simbólico BDI com conhecimento semântico posiciona Hypermedea como solução única para agentes em ambientes cyber-físicos semanticamente anotados (Charpenay et al., 2022).

3 DESCRIÇÃO DO PROBLEMA E REQUISITOS TÉCNICOS

Este capítulo apresenta uma descrição detalhada do problema abordado neste trabalho, bem como os requisitos técnicos que a solução proposta deve atender. A seção 3.1 apresenta o LAI, local de desenvolvimento do projeto, a seção 3.3 descreve o projeto “Agents in Charge”, que serviu como base para o desenvolvimento do presente trabalho, e a seção 3.4 apresenta as limitações e desafios encontrados na implementação do projeto, que motivaram a presente pesquisa.

3.1 AMBIENTE DO LABORATÓRIO LAI

Este trabalho foi desenvolvido no LAI, um laboratório de pesquisa da UFSC. O projeto utiliza uma bancada didática de manufatura da Festo, controlada por um CLP da Siemens, que integra diversos sensores e atuadores. A arquitetura de controle tradicional desta bancada, com lógica centralizada no CLP, resulta em um alto acoplamento entre software e hardware. O objetivo deste trabalho é usar essa plataforma para validar uma nova arquitetura baseada em WoT que abstrai a complexidade do hardware e melhora a interoperabilidade.

3.1.1 Descrição da Linha de Montagem

A linha de montagem utilizada neste trabalho simula um processo de manufatura de peças plásticas através de seis estações interconectadas: *Distributing*, *Testing*, *Separating*, *Pick and Place*, *Fluidic Muscle Press* e *Sorting*. Cada estação, ou bancada, é controlada por um CLP Siemens S7-1200, programado em linguagem Ladder. O fluxo de produção inicia-se na estação *Distributing*, que alimenta o sistema, e passa pela *Testing*, que verifica a altura das peças.

A comunicação entre bancadas é realizada através de pares sensor/receptor óticos que, assim como os demais sensores e atuadores, são controlados pelos CLPs de cada bancada. Cada estação monitora a disponibilidade da próxima antes de liberar peças, coordenando o fluxo de produção.

3.1.2 A Bancada Separating como Objeto de Estudo

Dentre as estações da linha, a bancada *Separating* foi selecionada como o objeto de estudo, servindo como base tanto para o projeto preliminar “Agents in Charge” quanto para o desenvolvimento do presente trabalho. A função principal desta bancada no processo produtivo é inspecionar a orientação das peças recebidas da estação de *Testing*. Peças que se apresentam invertidas são identificadas e descartadas, garantindo que apenas as peças com a orientação correta prossigam para a etapa seguinte, a *Pick and Place*, onde receberão uma tampa.

Para realizar esta função, a bancada é composta por um conjunto de sensores e atuadores, cuja disposição física é ilustrada no esquemático da Figura 1. O fluxo de uma peça através da bancada inicia-se com sua detecção pelo sensor de chegada (1 - *ArrivalSensor*). A peça é então transportada pela esteira principal (8 - *MainConveyor*) até a posição de inspeção, onde é detectada pelo sensor de peça parada (2 - *StoppedSensor*) e fixada pela trava (4 - *Lock*). Neste ponto, o sensor de altura (3 - *HeightSensor*) realiza a medição para inferir a orientação. Se a peça for considerada invertida, o atuador de descarte (6 - *DiscardDiverter*) é acionado para movê-la para a esteira de descarte (9 - *DiscardConveyor*), onde sua remoção é confirmada pelo sensor de descarte (5 - *DiscardSensor*). Caso contrário, a peça é liberada e continua na esteira principal até ser detectada pelo sensor de saída (7 - *ExitSensor*), sinalizando sua transferência para a próxima bancada.

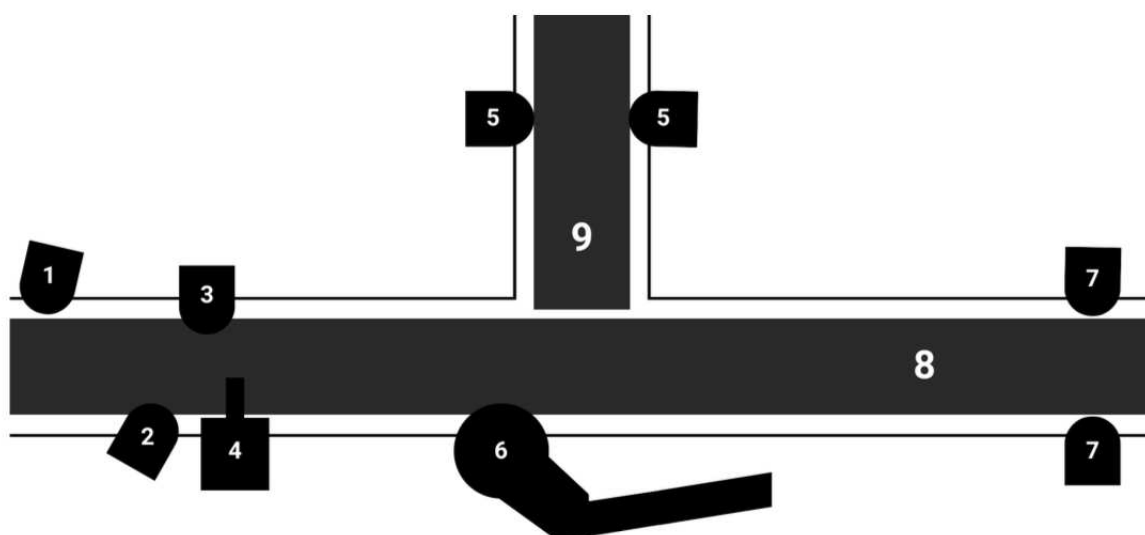


Figura 1 – Esquemático da bancada *Separating* com seus componentes numerados.

3.2 AMBIENTE DE SIMULAÇÃO PARA DESENVOLVIMENTO

Para permitir o desenvolvimento e a validação do projeto de forma desacoplada do hardware físico, foi criado um ambiente de simulação da planta industrial. Este simulador foi projetado com o objetivo de substituir o CLP, replicando fielmente seu comportamento e, crucialmente, apresentando a mesma interface *Open Platform Communications Unified Architecture* (OPC UA) da bancada FESTO real. Esta estratégia garante que qualquer aplicação cliente, projetada para se comunicar com o CLP físico, possa interagir com o simulador de forma idêntica, sem necessidade de qualquer alteração em seu código.

O simulador foi desenvolvido em *TypeScript*, utilizando a biblioteca *node-opcua* para a implementação do servidor OPC UA. Sua lógica interna opera em um ciclo de tempo discreto, com um intervalo de atualização de 10 milissegundos, e modela a bancada como um plano cartesiano bidimensional onde peças virtuais se movem. A implementação foi estruturada de forma orientada a objetos, com classes para representar as peças (*Piece*), os sensores (*Sensor*) e os atuadores (*Actuator*). O estado dos atuadores é controlado externamente através da interface OPC UA, e, a cada ciclo, o estado dos sensores é atualizado com base na posição das peças, que por sua vez é influenciada pelo estado dos atuadores.

Para facilitar o teste e a validação do sistema, o simulador expõe um *endpoint* HTTP que permite a inserção programática de peças na simulação, com a possibilidade de especificar sua orientação. Complementarmente, um sistema abrangente de *logs* foi implementado para registrar em detalhes o fluxo das peças, o estado dos sensores e as ações dos atuadores, fornecendo uma base sólida para a validação e depuração de qualquer sistema de controle desenvolvido sobre o simulador.

3.3 PROJETO “AGENTS IN CHARGE”

O problema central abordado neste trabalho tem origem no projeto “Agents in Charge”, desenvolvido previamente na disciplina de Projeto Integrador. Esse projeto serviu como a primeira iniciativa para conectar agentes inteligentes às bancadas didáticas do LAI, estabelecendo uma prova de conceito da integração entre sistemas multiagentes e automação industrial. Esta seção descreve os objetivos, a arquitetura e a implementação daquele projeto, fornecendo o contexto técnico necessário para a análise de suas limitações, que será detalhada na Seção 3.4

O objetivo principal do “Agents in Charge” era permitir que agentes inteligentes, desenvolvidos com base no modelo BDI (*Belief, Desire, Intention*), pudessem controlar remotamente as bancadas da FESTO. O escopo definiu a criação de um middleware com uma API REST para expor as funcionalidades dos sensores e atuadores, além da adoção do padrão WoT para a descrição formal dos dispositivos. Como resultado, foi entregue uma arquitetura funcional, focada na bancada *Separating* e encapsulada em um contêiner Docker, que validou a viabilidade da comunicação entre agentes e o ambiente físico. Apesar de funcional, a abordagem revelou desafios inerentes de acoplamento e escalabilidade, que motivaram a presente pesquisa.

3.3.1 Arquitetura do Sistema

A arquitetura do projeto foi estruturada em três camadas distintas para abstrair a complexidade do hardware e padronizar a interação. A camada física era composta pela **Bancada FESTO**, controlada por um CLP Siemens S7-1200. A camada de in-

teligência era representada pelo **Agente Inteligente**, responsável pela tomada de decisões. Essas duas camadas são conectadas por meio da camada de integração, formada por um **Middleware baseado em Node-RED**, que orquestrava a comunicação.

O fluxo de informações operava da seguinte maneira: o agente enviava requisições HTTP para a API REST exposta pelo middleware, utilizando o método GET para consultar o estado de sensores e POST para invocar ações em atuadores. O Node-RED, atuando como um cliente OPC UA, recebia essas requisições, as traduzia em comandos para o CLP e enviava a resposta do hardware de volta ao agente.

3.3.2 Middleware Baseado em Node-RED

O middleware, implementado na plataforma Node-RED, atuou como componente central da arquitetura de integração. Para estabelecer a comunicação com o CLP, utilizou-se a biblioteca `node-red-contrib-opcua`, que permitia a interação com o servidor OPC UA em execução no controlador. A integração foi viabilizada através de um programa em linguagem Ladder desenvolvido no TIA Portal e carregado no CLP. Este programa realizava o mapeamento das entradas e saídas físicas da bancada para um *Data Block* específico, cujas variáveis eram então expostas através do protocolo OPC UA.

Os fluxos em Node-RED foram responsáveis por implementar a lógica da API, incluindo a definição de *endpoints* para cada dispositivo (sensores e atuadores), a validação de tipos de dados nas requisições e a verificação de integridade para garantir que o programa correto estava em execução no CLP. A organização do middleware foi otimizada com o uso de subfluxos para encapsular operações repetitivas, como as requisições GET e POST.

3.3.3 Implementação das TD

Com o objetivo de promover a interoperabilidade, o projeto “Agents in Charge” adotou o padrão WoT para a descrição dos dispositivos. Foi criado um documento JSON, seguindo os princípios do TD, para a bancada *Separating*, formalizando seus *affordances*. A estrutura da TD seguiu o padrão da W3C, utilizando **properties** para representar os sensores e o estado dos atuadores, e **actions** para modelar as operações que poderiam ser executadas.

O mecanismo de comunicação foi explicitado no campo `forms`, que detalhava para cada *property* e *action* o método HTTP a ser utilizado e o `href` com a URL do *endpoint* na API. Essa abordagem, embora funcional, limitava-se a descrever como interagir com os dispositivos, mas não fornecia um modelo de conhecimento que representasse o porquê de suas operações de forma processável por máquina. Faltava um vocabulário semântico que permitisse a um agente autônomo, por exemplo, inferir

que a leitura do `height_sensor` servia para verificar a orientação da peça, e não sua altura máxima, ou qual a relação funcional entre este sensor e o atuador de descarte. Essa ausência de uma camada de significado explícito e interpretável por máquina é uma das limitações centrais que serão discutidas na seção seguinte.

3.4 LIMITAÇÕES E DESAFIOS ATUAIS

O projeto “Agents in Charge”, apesar de apresentar uma arquitetura funcional, principalmente no que diz respeito ao Middleware WoT, demonstrou desafios significativos na implementação da camada de inteligência. A integração dos agentes inteligentes com o sistema foi realizada apenas de forma prototípica, sem a adoção de uma abordagem mais robusta para a modelagem dos agentes, como os agentes BDI.

O agente foi desenvolvido em Python, e foi programado com a lógica exata esperada para a bancada *Separating*. Embora o agente tenha utilizado a TD para descobrir dinamicamente os endpoints de comunicação, os nomes dos componentes são acessados diretamente no código, como é possível notar no Código 3.1, ao invés de serem acessados através de seus significados semânticos (classes). Portanto, para a expansão do projeto para outras bancadas, seria necessário programar manualmente a lógica e interações REST com cada componente do Middleware, o que seria uma tarefa complexa e propensa a erros.

As próximas subseções apresentam detalhadamente as limitações relacionadas ao acoplamento entre agentes e dispositivos, à falta de compreensão semântica e à escalabilidade e reusabilidade limitadas.

3.4.1 Alto Acoplamento Entre Agentes e Dispositivos

A implementação atual do agente em Python, embora utilize a Thing Description para obter os endereços dos *endpoints*, exhibe um alto grau de acoplamento lógico e nominal com os dispositivos da bancada *Separating*. Este acoplamento manifesta-se na forma como o agente é programado para interagir com os componentes através de seus nomes estáticos, como `arrival_sensor` ou `activate_lock`, que são utilizados como chaves fixas no código para localizar as informações de comunicação dentro do documento TD.

O alto acoplamento, portanto, não reside no endereço final do *endpoint*, que é de fato descoberto dinamicamente, mas sim na dependência do agente em relação aos identificadores nominais e à estrutura interna da TD. O código do agente está intrinsecamente ligado a chaves estáticas como `arrival_sensor`, o que o torna frágil a qualquer refatoração ou evolução no nome dos componentes descritos na TD, mesmo que a funcionalidade do dispositivo permaneça idêntica.

Conforme demonstrado no Código 3.1, extraído da implementação do agente

Python, as URLs dos componentes são obtidas das TDs e armazenadas em variáveis estáticas:

```

1 discard_diverter_root = data['properties']['discard_diverter_status']['forms'][0]['href']
2 arrival_sensor_root = data['properties']['arrival_sensor']['forms'][0]['href']
3 stopped_sensor_root = data['properties']['stopped_sensor']['forms'][0]['href']
4 main_conveyor_root = data['actions']['activate_main_conveyor']['forms'][0]['href']
5 lock_root = data['actions']['activate_lock']['forms'][0]['href']

```

Código 3.1 – Extração das URLs dos componentes a partir das TDs

Embora o agente faça uso das TDs para descobrir dinamicamente os endpoints, toda a lógica de controle permanece codificada de forma rígida no código Python. O conhecimento sobre o comportamento operacional da bancada, incluindo as sequências de ativação, condições de decisão e tratamento de estados, está totalmente incorporado no agente, como evidenciado no Código 3.2:

```

1 if get_func(arrival_sensor_root):
2     print('Ativando trava...')
3     post_func(lock_root, False)
4     print('Ativando esteira principal...')
5     post_func(main_conveyor_root, True)
6
7 while True:
8     if get_func(stopped_sensor_root):
9         if get_func(height_sensor_root):
10            print('Altura incorreta detectada. Encaminhando para descarte...')
11            post_func(discard_conveyor_root, True)
12            post_func(discard_diverter_root, True)

```

Código 3.2 – Lógica de controle codificada estaticamente no agente

Esta abordagem implica que qualquer alteração na lógica operacional ou na configuração dos dispositivos requer modificações manuais no código fonte, comprometendo significativamente a flexibilidade do sistema.

3.4.2 Falta de Compreensão Semântica

As TDs expostas pelo projeto “Agents in Charge” carecem de anotações semânticas que permitam aos agentes compreender o significado e as relações entre os dispositivos de forma autônoma. O documento TD gerado, embora siga o padrão WoT, descreve apenas a estrutura da comunicação, como o método HTTP e a URL de um *endpoint*, sem atribuir um significado formal e processável por máquina às operações.

O Código 3.3 exemplifica a descrição específica do sensor de altura utilizada no projeto para ilustrar esta limitação.

```
1 "height_sensor": {
2   "title": "Height sensor.",
3   "description": "Returns the height (upside-down part).",
4   "objectType": "sensor",
5   "type": "object",
6   "properties": {
7     "value": "boolean"
8   },
9   "readOnly": true,
10  "forms": [
11    {
12      "href": "/api/separating/heightSensor",
13      "contentType": "application/json",
14      "op": "readproperty",
15      "htv:methodName": "GET"
16    }
17  ]
18 }
```

Código 3.3 – Exemplo de TD sem anotações semânticas formais

Observa-se que, embora a descrição textual mencione “upside-down part”, não há anotações formais que permitam a um agente inferir automaticamente que este sensor está relacionado a uma função de verificação de orientação, ou que seu acionamento deve desencadear uma sequência de descarte. A implementação atual do agente Python evidencia essa lacuna, pois a lógica de interpretação do valor do sensor é codificada manualmente pelo desenvolvedor, como demonstra o Código 3.4.

```
1 if get_func(height_sensor_root): # Desenvolvedor interpreta "True" como
   peça com altura errada
2   print('liga a esteira de descarte')
3   post_func(discard_conveyor_root, True)
4   print('ativa o separador de descarte')
5   post_func(discard_diverter_root, True)
```

Código 3.4 – Interpretação manual da semântica dos sensores

A estrutura do código, com o bloco condicional que se segue à leitura do sensor, demonstra que o desenvolvedor teve que interpretar o significado do valor booleano e codificar a relação causal entre a ativação do `height_sensor` e a necessidade de acionar os componentes de descarte. Essa lógica de negócio, portanto, reside inteiramente no agente, de forma explícita e inflexível. Similarmente, a TD do atuador de descarte, mostrada no Código 3.5, descreve apenas os detalhes técnicos para a sua invocação, como o método HTTP e a URL, sem fornecer um modelo de suas capacidades.

```
1 "activate_discard_diverter": {
2   "title": "Discard diverter for part.",
3   "description": "Changes the state of the discard claw for part
4     redirection.",
5   "input": {
6     "type": "object",
7     "properties": {
8       "value": "boolean"
9     }
10  },
11  "forms": [
12    {
13      "href": "/api/separating/discardDiverter",
14      "contentType": "application/json",
15      "op": "invokeaction",
16      "htv:methodName": "POST"
17    }
18  ]
19 }
```

Código 3.5 – TD de atuador sem relações semânticas

Não há indicações semânticas que estabeleçam relações como este atuador deve ser acionado quando peças defeituosas são detectadas” ou este componente pertence ao subsistema de controle de qualidade”. Esta limitação é particularmente crítica para agentes BDI, cuja arquitetura se baseia na premissa de que as crenças (*beliefs*) sobre o mundo devem ser formadas a partir de percepções do ambiente. Uma descrição ontológica serviria como uma fonte de percepção rica e autoritativa, permitindo ao agente derivar conhecimento sobre o domínio em tempo de execução, ao invés de depender de lógica estaticamente codificada.

3.4.3 Escalabilidade e Reusabilidade Limitadas

A implementação atual evidencia limitações quanto à escalabilidade para novas bancadas e à reusabilidade dos componentes desenvolvidos. A ausência de implementação para outras bancadas do sistema FESTO não decorre apenas de limitações de escopo, mas reflete a complexidade inerente à abordagem atual, onde cada nova bancada demandaria o desenvolvimento de um agente com lógica de controle específica e programada manualmente.

O agente em Python cumpriu seu papel de consumir a API exposta pelo middleware e controlar a bancada, demonstrando a viabilidade da comunicação via TD. No entanto, quando se considera a relação esforço-benefício para a expansão do sistema, esta abordagem prototípica não apresenta vantagens claras sobre métodos de automação convencionais.

A vantagem de uma arquitetura semântica, como a proposta neste trabalho, não reside necessariamente na eliminação de um agente por bancada, mas sim na transformação da natureza desse agente. Em vez de ser programado para interagir com dispositivos através de seus identificadores nominais (e.g., “height_sensor”), o agente passa a operar com base em funcionalidades semânticas (e.g., “encontrar um sensor que realiza a função de verificação de orientação”). Com uma fonte de conhecimento semântico, a lógica do agente torna-se robusta a mudanças na infraestrutura física. A substituição de um sensor ou atuador por um modelo diferente, com outro nome ou interface, exigiria apenas uma atualização no Grafo de Conhecimento. O agente, ao consultar o grafo pela funcionalidade desejada, descobriria o novo componente dinamicamente, sem que nenhuma alteração em seu código fosse necessária. Isso representa um ganho substancial em reusabilidade e uma drástica redução no esforço de manutenção e reconfiguração do sistema.

O desafio fundamental da arquitetura anterior reside, portanto, na necessidade de codificação manual de todo o conhecimento operacional da bancada em cada agente. Mesmo utilizando TD como interface de comunicação, o conhecimento sobre comportamentos, sequências de controle e lógicas de decisão deve ser explicitamente programado, impedindo que a promessa de desacoplamento e reutilização inerente à arquitetura WoT se materialize plenamente na prática.

É importante ressaltar que a crítica se concentra na camada de inteligência, e não na infraestrutura WoT subjacente, cujos benefícios serão mantidos na solução proposta. A exposição dos dispositivos via protocolo HTTP e a padronização via TD facilitam a interoperabilidade, a documentação automática e a integração com outras ferramentas, como dashboards de monitoramento, sendo uma base sólida sobre a qual a nova camada semântica será construída.

3.5 REQUISITOS TÉCNICOS

Com base na análise das limitações do projeto “Agents in Charge” e nos objetivos estabelecidos para este trabalho, foram identificados os requisitos técnicos necessários para o desenvolvimento de uma solução baseada em grafos de conhecimento e agentes BDI semânticos.

3.5.1 Requisitos Funcionais

Os requisitos funcionais definem as funcionalidades que o sistema deve implementar para atender aos objetivos propostos:

RF01 — Modelagem Semântica da Planta: O sistema deve representar a bancada *Separating* e seus componentes como um Grafo de Conhecimento em RDF. Este modelo deve incluir a estrutura hierárquica da planta (e.g., bancada contém subsis-

temas, que contêm dispositivos) e as capacidades funcionais de cada componente, utilizando ontologias padrão.

RF02 — Exposição do Grafo de Conhecimento: O sistema deve fornecer um servidor HTTP que exponha o Grafo de Conhecimento. Este servidor deve suportar a recuperação de descrições completas ou parciais do grafo e oferecer os dados em múltiplos formatos de serialização RDF via negociação de conteúdo (*content negotiation*).

RF03 — Descoberta Semântica de Capacidades: O agente de controle deve ser capaz de navegar pelo grafo, partindo de uma URI base da bancada, para descobrir dinamicamente os dispositivos disponíveis e suas capacidades funcionais (e.g., “sensoriamento de orientação”, “atuação de descarte”).

RF04 — Execução de Controle Baseada em Semântica: Após descobrir um dispositivo com a capacidade desejada, o agente deve ser capaz de extrair do grafo as informações de interação (i.e., as *affordances* da TD) e utilizá-las para executar operações de controle, como ler o estado de um sensor ou invocar a ação de um atuador.

3.5.2 Requisitos Não-Funcionais

Os requisitos não-funcionais especificam características de qualidade do sistema:

RNF01 — Escalabilidade: A arquitetura do Grafo de Conhecimento deve ser projetada de forma modular e hierárquica, permitindo a futura expansão para modelar outras bancadas da linha de produção FESTO com mínimo impacto na estrutura existente.

RNF02 — Manutenibilidade: Alterações na infraestrutura física (e.g., substituição de um sensor por um modelo diferente) devem ser refletidas apenas por meio de atualizações no Grafo de Conhecimento, sem a necessidade de modificar ou recompilar o código do agente de controle.

3.5.3 Restrições de Projeto

As restrições de projeto definem condições ou tecnologias que devem ser obrigatoriamente utilizadas:

RP01 - Utilização de Vocabulários Padronizados: A solução deve, obrigatoriamente, priorizar o uso de ontologias consolidadas e recomendadas pela indústria, como SOSA e SAREF, para garantir a interoperabilidade. Extensões customizadas devem ser criadas apenas para conceitos não cobertos pelos padrões.

RP02 - Compatibilidade com a Infraestrutura Existente: A solução deve manter total compatibilidade com o middleware WoT baseado em Node-RED já existente,

consumindo a API REST por ele exposta, sem demandar modificações em sua implementação.

RP03 - Utilização da Plataforma de Agentes JaCaMo: O agente de controle deve ser desenvolvido utilizando a plataforma JaCaMo, e o framework Hypermedea para o consumo dos dados em RDF, eliminando a necessidade de implementar um cliente SPARQL no agente.

RP04 - Simplicidade de Segurança: Dado o contexto de um ambiente de laboratório controlado, o sistema não deve implementar mecanismos complexos de autenticação ou autorização.

4 SOLUÇÃO PROPOSTA E METODOLOGIA UTILIZADA

Este capítulo apresenta a solução proposta para os problemas identificados anteriormente, bem como a metodologia utilizada para o desenvolvimento e implementação da solução.

4.1 VISÃO GERAL DA SOLUÇÃO PROPOSTA

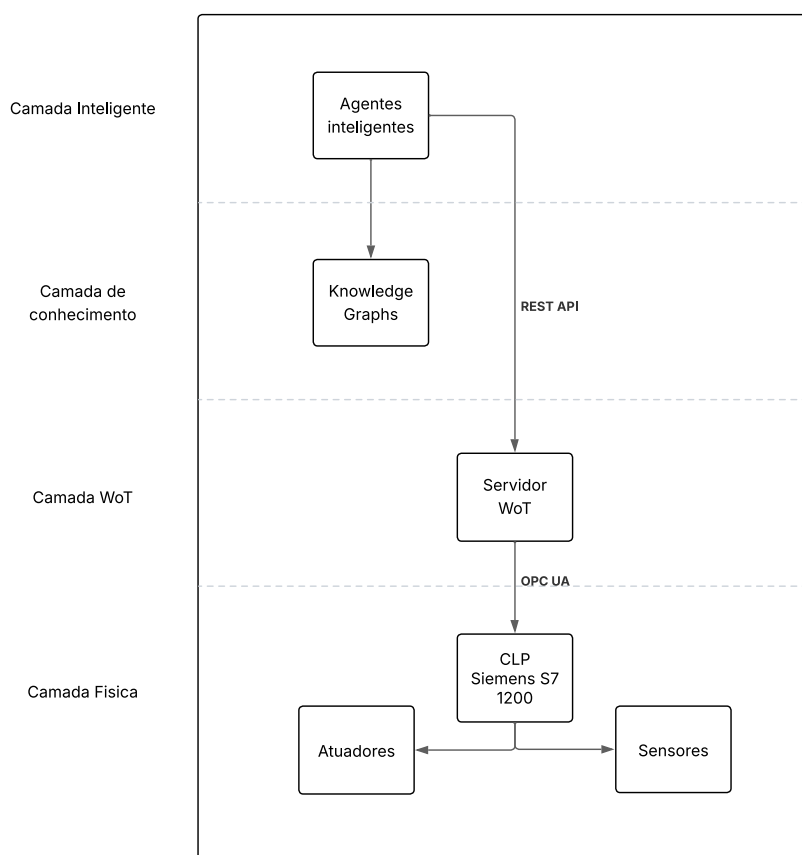
Para superar as limitações de acoplamento identificadas no projeto “Agents in Charge”, este trabalho propõe uma arquitetura que complementa a infraestrutura WoT existente, introduzindo uma camada de conhecimento semântico. O conceito fundamental consiste em externalizar o conhecimento operacional, antes codificado estaticamente no agente, para um Grafo de Conhecimento em RDF. Este grafo atua como uma fonte única e autoritativa de informação, representando não apenas os detalhes de interação que estariam em uma TD (as *affordances*), mas, crucialmente, as capacidades funcionais dos dispositivos e suas relações hierárquicas. Dessa forma, o agente deixa de depender de identificadores nominais para descobrir componentes, passando a realizar consultas semânticas por suas funções. A solução visa, portanto, reduzir o acoplamento entre o agente e a planta industrial, permitindo que a lógica de controle se torne robusta a mudanças na infraestrutura.

A arquitetura proposta para alcançar este desacoplamento é estruturada em quatro camadas distintas e interconectadas, que separam as responsabilidades desde a interação física até a tomada de decisão autônoma, conforme ilustrado na Figura 2. Na base, a camada física compreende o hardware da planta FESTO, e a camada de WoT atua como sua interface digital, expondo as funcionalidades dos dispositivos através de uma API REST, mantendo assim a infraestrutura de comunicação já validada no projeto anterior.

O componente inovador da solução reside nas duas camadas superiores. A camada de conhecimento é responsável por manter e expor o Grafo de Conhecimento em RDF. Este grafo descreve semanticamente a planta, suas capacidades funcionais e as informações de interação, servindo como uma fonte única da verdade sobre o ambiente. No topo, a camada inteligente contém o agente BDI, onde a lógica sequencial de controle estará programada, tomando como base as informações de ambiente obtidas.

Nesta arquitetura, para satisfazer um objetivo, o agente na camada inteligente inicia um ciclo de descoberta ao consultar a camada de conhecimento. Primeiramente, o agente é programado com um plano de controle a ser seguido. A partir do plano, ele busca por dispositivos que possuam a capacidade funcional necessária e extrai os detalhes de sua interface de interação. De posse dessas informações, o agente então envia uma requisição HTTP diretamente para a camada WoT, que por sua vez

Figura 2 – Arquitetura da Solução Proposta



a traduz em um comando para a camada física. O resultado da operação retorna pelo mesmo caminho, permitindo que o agente atualize suas crenças e continue seu ciclo deliberativo, completando assim o controle do processo.

4.2 METODOLOGIA DE PROJETO

A metodologia adotada neste trabalho foi estruturada para refletir uma colaboração técnica com um desenvolvedor especialista em agentes autônomos. Nesta dinâmica, este projeto atua como fornecedor de uma estrutura semântica, enquanto o colaborador atua como consumidor, utilizando esta estrutura para o desenvolvimento do agente de controle. Essa dinâmica impõe a necessidade de uma interface clara e bem definida entre as partes. Diferentemente de uma API de software tradicional, a interface neste contexto é, por sua natureza semântica, o próprio vocabulário compartilhado. Este vocabulário dita tanto a estrutura e as regras de construção do Grafo de Conhecimento quanto a lógica de consulta que o Agente deve utilizar para interpretá-lo.

4.2.1 Princípios de Design para a Modelagem Semântica

Antes da seleção de ontologias, a metodologia foi guiada por um conjunto de princípios de design para garantir que o Grafo de Conhecimento resultante fosse robusto, interoperável e de alta qualidade. O princípio fundamental foi o da **reutilização** de padrões, evitando a criação de vocabulários customizados sempre que um padrão da indústria ou da comunidade estivesse disponível. Adicionalmente, o princípio da **modularidade** foi aplicado, buscando-se a composição de múltiplas ontologias ortogonais, cada uma responsável por uma faceta da descrição (estrutura, função, interação). Por fim, o princípio da **especialização contextual** guiou a extensão dos padrões, onde conceitos genéricos (como `sosa:Sensor`) foram especializados com classes de domínio mais específicas (como `ai4i:PresenceSensor`) para aumentar a expressividade do modelo sem sacrificar a conformidade com os padrões de base.

4.2.2 Definição da Interface Semântica via Ontologias Padrão

O pilar da metodologia é a definição explícita da Interface Semântica, que serve como o vocabulário fundamental para a comunicação entre o Grafo de Conhecimento e o agente. A premissa é que tanto o fornecedor (ao modelar a planta) quanto o consumidor (ao programar o agente) devem aderir a este conjunto de ontologias e vocabulários. Esta interface foi construída através da integração de múltiplos vocabulários padrão, onde cada um contribui com uma camada específica de significado, resultando em um modelo capaz de descrever um sistema de automação industrial de forma abrangente.

A base da descrição é formada pela ontologia SOSA/SSN, escolhida por sua robustez como padrão da W3C para a modelagem da estrutura física e do processo de observação (Haller et al., 2017). Seu papel na arquitetura é fornecer um vocabulário formal para representar a composição hierárquica de sistemas complexos. Através de seus conceitos, é possível decompor uma planta industrial em sistemas, subsistemas e, finalmente, em dispositivos individuais como sensores e atuadores. A SOSA permite modelar não apenas a existência desses componentes, mas também suas inter-relações físicas, como a relação de um dispositivo estar hospedado (`sosa:hosts`) em uma plataforma, fornecendo a espinha dorsal estrutural do Grafo de Conhecimento.

Para descrever o propósito e a funcionalidade dos componentes, a ontologia SAREF foi incorporada à interface (ETSI, 2022). Enquanto a SOSA responde à pergunta “o que este dispositivo é?”, a SAREF responde à pergunta “o que este dispositivo faz?”. Esta camada de significado funcional é crucial para o desacoplamento, pois habilita a descoberta de recursos baseada em capacidades, e não em identificadores. A SAREF oferece um modelo explícito para representar as Funções que um dispositivo pode realizar (`saref:accomplishes`), os Comandos que podem ser enviados e os Estados que podem ser assumidos, fornecendo ao consumidor do grafo o contexto semântico

necessário para tomar decisões informadas sobre quais recursos utilizar para atingir seus objetivos.

Embora as ontologias SOSA e SAREF forneçam uma base sólida e genérica, sistemas industriais frequentemente possuem conceitos de domínio muito específicos. Para evitar a criação de um vocabulário customizado extenso e pouco interoperável, a metodologia incluiu a reutilização de ontologias de domínio já existentes, como a ontologia definida pelo AI4Industry (Nardin, 2024). O papel dessa ontologia é fornecer especializações de conceitos, como classes mais específicas de sensores ou propriedades para descrever zonas funcionais (e.g., áreas de entrada e saída), que não são cobertas pelos vocabulários de base. Esta abordagem de estender padrões com vocabulários de domínio validados é uma prática recomendada que aumenta a expressividade do modelo sem perder a interoperabilidade.

Finalmente, para conectar o modelo semântico abstrato à camada de interação concreta, a ontologia TD foi integrada, semanticamente mantendo a compatibilidade com o padrão WoT (Käbisch et al., 2023). Enquanto as outras ontologias descrevem a estrutura e a função, a TD descreve como interagir com cada dispositivo no nível do protocolo. Seu papel é encapsular os detalhes práticos da comunicação, como o método HTTP e a URL de um *endpoint* dentro de *affordances*. Ao anotar cada dispositivo no grafo com suas respectivas *affordances*, a interface garante que, uma vez que o agente tenha descoberto um dispositivo e entendido sua função, ele também tenha toda a informação necessária para efetivamente se comunicar com ele, completando o ciclo de descoberta e ação.

4.2.3 Procedimentos de Verificação e Validação

Para assegurar que a solução proposta atende aos seus objetivos e requisitos, a metodologia inclui um conjunto de procedimentos de verificação e validação. A etapa de verificação busca compreender se o produto atende aos requisitos, já a validação, se o produto resolve o problema apresentado e contextualizado no Capítulo 3.

Para a verificação, será realizado um mapeamento direto entre cada requisito funcional e não-funcional definido na Seção 3.5 e a implementação correspondente. Por exemplo, o requisito de Descoberta Semântica (RF03) será verificado observando a capacidade que o agente construído possui de encontrar um dispositivo com base em sua função, sem conhecimento prévio de seu identificador.

Para a validação, a metodologia propõe um cenário de teste qualitativo focado no problema central do desacoplamento. Este cenário simulará uma alteração na infraestrutura de hardware, onde um sensor no Grafo de Conhecimento terá seu identificador e URL de endpoint modificados, mantendo, no entanto, sua descrição funcional semântica.

O critério fundamental para a validação da solução proposta será a capacidade

do agente de controle, sem qualquer alteração em seu código-fonte, de se adaptar automaticamente à mudança estrutural e continuar operando a bancada corretamente. Este comportamento demonstrará empiricamente que a arquitetura semântica efetivamente reduz o acoplamento entre o agente e a infraestrutura física.

4.3 ARQUITETURA DA SOLUÇÃO E TECNOLOGIAS PROPOSTAS

Nesta seção, será apresentada a arquitetura detalhada da camada de conhecimento, foco deste trabalho. A materialização dessa camada é o passo mais crítico desta arquitetura, pois é nela que reside a inteligência semântica do sistema.

A decisão de como construir, armazenar e expor o Grafo de Conhecimento impacta diretamente a manutenibilidade, a escalabilidade e a performance da solução. Durante o projeto, foram consideradas diversas abordagens para a implementação desta camada, e a escolha final foi resultado de uma análise comparativa de suas vantagens e desvantagens no contexto desta aplicação.

4.3.1 Análise de Abordagens para a Implementação do Grafo

Para a implementação da camada de conhecimento, quatro arquiteturas principais foram avaliadas. Para a comparação das diferentes abordagens, foram definidos três critérios relevantes para o projeto, que são:

- **Manutenibilidade do modelo:** A capacidade e facilidade de manter e atualizar o modelo sem comprometer a qualidade e a coerência do conhecimento.
- **Suporte à exposição em múltiplos formatos:** O suporte a diferentes formatos de serialização do Grafo de Conhecimento, como Turtle, JSON-LD, RDF/XML, etc.
- **Escalabilidade futura:** A capacidade de suportar o crescimento do Grafo de Conhecimento e o aumento da complexidade das consultas, bem como a implementação de novas funcionalidades.

A Tabela 2 resume a análise comparativa das diferentes abordagens, considerando os três critérios definidos.

4.3.1.1 Arquivo RDF Estático

A primeira abordagem considerada foi a de armazenar o Grafo de Conhecimento em um arquivo RDF estático. Esta abordagem é a mais simples e direta, mas apresenta várias desvantagens. A principal delas é a dificuldade de manutenção e evolução do modelo, pois a edição manual de Turtle é propensa a erros e difícil para modelos complexos. Além disso, o suporte à exposição seria a criação de um servidor que

Tabela 2 – Análise comparativa de abordagens para a camada de conhecimento.

Abordagem	Manutenibilidade	Suporte à Exposição	Escalabilidade
Arquivo RDF Estático	Baixa	Limitado	Baixa
JSON-LD Estático (TD Expandido)	Média	Limitado	Baixa
Banco Relacional + Conversão	Alta	Flexível	Média
Jena com TDB2 (Adotada)	Alta	Flexível	Alta

Fonte: Elaborado pelo autor.

apenas expõe um arquivo, prejudicando a navegação pelas triplas RDF e a descoberta de recursos, e não suportando diferentes formatos de serialização.

Finalmente, a arquitetura de arquivo estático é inerentemente não escalável. Como o servidor nesta abordagem se limita a entregar o conteúdo de um arquivo, ele não executa qualquer lógica de processamento ou consulta sobre os dados. Consequentemente, ele não suporta mecanismos de interação do lado do servidor, como um *endpoint* SPARQL. Esta limitação arquitetural impõe que qualquer agente consumidor deva sempre requisitar e processar o grafo em sua totalidade. Para um sistema projetado para crescer e abranger múltiplas bancadas, esta abordagem se tornaria um gargalo de performance e consumo de recursos, sendo, portanto, inadequada para os objetivos de longo prazo do projeto.

4.3.1.2 JSON-LD Estático (TD Expandido)

A segunda abordagem considerada foi a de armazenar o Grafo de Conhecimento em um arquivo JSON-LD estático. Esta abordagem se baseia na expansão de uma única TD para conter não apenas as *affordances*, mas também as anotações semânticas de toda a planta. O seu principal atrativo é a possibilidade de utilizar o mecanismo `@context` do JSON-LD para mapear termos simples a conceitos de ontologias como SOSA e SAREF, mantendo a familiaridade e a legibilidade do formato JSON. Em termos de manutenibilidade, isso representa uma vantagem sobre a edição manual de Turtle. No entanto, esta abordagem compartilha desvantagens críticas com a arquitetura de arquivo RDF estático, notadamente a baixa escalabilidade para consultas complexas e a dificuldade na exposição em múltiplos formatos de serialização.

O fator decisivo para a rejeição desta abordagem, contudo, reside no seu desalinhamento com a metodologia de consumo de dados do agente. O framework Hypermedea, adotado para o desenvolvimento do agente BDI, é projetado para operar em um ambiente Linked Data, onde ele descobre o mundo ao navegar entre recursos através de suas URIs e das relações que os conectam. Um modelo de conhecimento contido em um único e monolítico documento JSON-LD não se adequa a este paradigma de navegação, pois o agente teria dificuldades para explorar os subsistemas e

componentes de forma granular, quebrando o ciclo de descoberta que é central para a solução proposta.

4.3.1.3 Banco Relacional e Conversão para RDF

A terceira abordagem avaliada foi a utilização de um banco de dados relacional tradicional, como o *PostgreSQL*, gerenciado por um framework web como o *Ruby on Rails*. Nesta arquitetura, a estrutura da planta seria modelada em um esquema de tabelas, onde, por exemplo, uma tabela de “dispositivos” poderia conter colunas que representam propriedades como *sosa:hosts* ou *saref:accomplishes*. Uma camada de conversão seria responsável por traduzir os dados relacionais para o formato RDF em tempo de execução, no momento da requisição ao servidor. A principal vantagem desta arquitetura reside na alta manutenibilidade do modelo de dados subjacente, aproveitando a maturidade e a familiaridade das ferramentas do ecossistema SQL. Além disso, a camada de conversão ofereceria grande flexibilidade para expor o grafo em múltiplos formatos de serialização, como Turtle ou JSON-LD, de acordo com a necessidade do cliente.

A sua principal desvantagem, no entanto, é o desalinhamento fundamental entre o paradigma relacional, que é rígido e tabular, e a natureza flexível e heterogênea de um grafo de conhecimento. A representação de relações complexas ou de componentes com conjuntos de propriedades variados torna-se um desafio, frequentemente resultando em esquemas de banco de dados com muitas colunas nulas ou em uma lógica de mapeamento excessivamente complexa. Concluiu-se que esta abordagem não elimina a complexidade, mas apenas a transfere para a camada de conversão, que se tornaria um gargalo de performance e um ponto crítico de manutenção, contrariando os objetivos de simplicidade e robustez da solução proposta.

4.3.1.4 Apache Jena com TDB2

A quarta e última abordagem analisada, que foi a adotada para este projeto, é a utilização de um *Triple Store*, um sistema de banco de dados nativo para grafos de conhecimento. Para implementar esta arquitetura, a tecnologia escolhida foi o framework de código aberto Apache Jena, em conjunto com seu motor de persistência de alto desempenho, o TDB2. O Apache Jena é um dos ecossistemas de ferramentas mais robustos e amplamente utilizados para a construção de aplicações para a Web Semântica na plataforma Java. Ele oferece um conjunto completo de funcionalidades para criar, manipular, consultar e persistir modelos RDF, tornando-se a solução ideal para as necessidades deste projeto.

A adoção desta abordagem resolve diretamente as limitações identificadas nas alternativas anteriores. Primeiramente, a manutenibilidade do grafo é muito alta, pois o Jena oferece uma API programática que permite a construção do modelo de forma

modular e orientada a objetos, como será detalhado no Capítulo 5. Isso elimina a necessidade de edição manual de arquivos estáticos, reduzindo a propensão a erros e facilitando a gestão de um grafo complexo.

Em segundo lugar, o suporte à exposição é nativo e flexível. O Jena possui serializadores integrados para todos os principais formatos RDF, como Turtle, JSON-LD e RDF/XML. Isso simplifica a implementação do servidor que exponhará o grafo, permitindo que ele atenda ao requisito de negociação de conteúdo de forma eficiente, sem a necessidade de uma camada de conversão customizada.

Finalmente, a escalabilidade da solução é garantida pelo uso do TDB2, um motor de armazenamento otimizado para grandes volumes de dados. Embora o agente deste projeto consuma o grafo através da navegação em arquivos RDF expostos via HTTP, a escolha pelo Apache Jena oferece um caminho claro para futuras evoluções. O ecossistema Jena inclui ferramentas como o servidor Fuseki, que pode expor o mesmo grafo TDB2 através de um *endpoint* SPARQL completo. Isso permite que futuras versões do agente, ou outros sistemas consumidores, realizem consultas complexas e diretas sobre o grafo. Portanto, a combinação do Jena com o TDB2 não apenas atende aos requisitos atuais do projeto, mas também fornece uma base robusta e preparada para a expansão das capacidades de interação com o Grafo de Conhecimento.

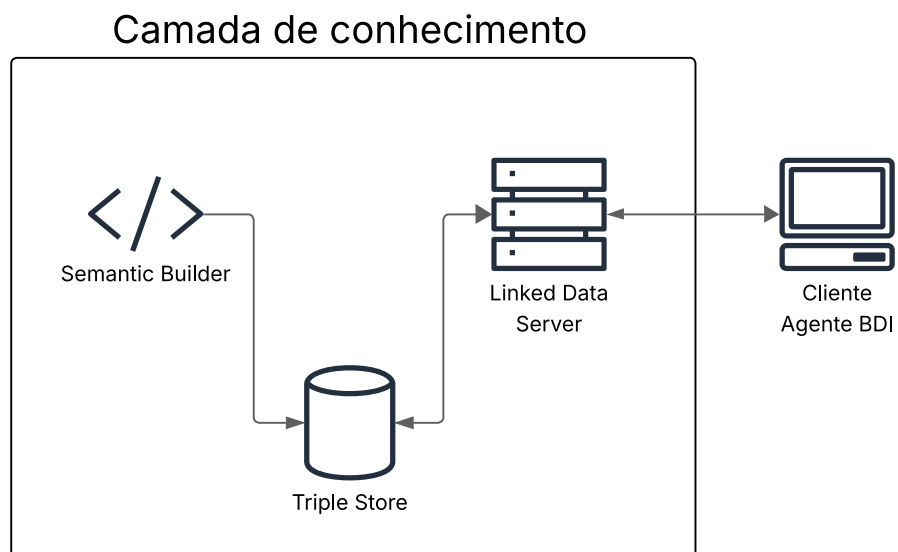
4.3.2 Arquitetura da Camada de Conhecimento

A implementação da camada de conhecimento, utilizando o framework Apache Jena, foi projetada de forma modular, separando as responsabilidades de construção e de exposição do grafo. A Figura 3 ilustra os dois componentes lógicos principais desta arquitetura: o Construtor Semântico (*Semantic Builder*) e o Servidor de *Linked Data* (*Linked Data Server*).

O Construtor Semântico é o componente responsável por criar o Grafo de Conhecimento. Sua função é instanciar as representações dos componentes da planta, como as bancadas, subsistemas e dispositivos, e serializá-los em triplas RDF de acordo com as ontologias definidas na Interface Semântica. A decisão de isolar esta lógica em um componente separado garante que a construção do grafo seja um processo controlado, consistente e de fácil manutenção, realizado através de uma API programática em Java.

O Servidor de *Linked Data*, por sua vez, é o componente responsável por expor o grafo armazenado no TDB2 para o mundo externo. Sua função é receber requisições HTTP, ler os dados solicitados do banco de dados e servi-los no formato RDF apropriado, de acordo com a negociação de conteúdo. Esta separação de responsabilidades garante que a lógica de exposição seja desacoplada da lógica de construção, permitindo, por exemplo, que o servidor seja atualizado ou substituído no futuro sem impactar a forma como o grafo é gerado. Os detalhes da implementação concreta

Figura 3 – Arquitetura da Camada de Conhecimento



Fonte: do autor.

destes componentes serão apresentados no Capítulo 5.

5 DESENVOLVIMENTO E ANÁLISE DOS RESULTADOS OBTIDOS

Neste capítulo, serão apresentados os detalhes da implementação da solução proposta. A seção 5.1 aborda a modelagem semântica da bancada Separating. A seção 5.2 apresenta a implementação da camada de conhecimento com Apache Jena. A seção 5.3 explica a infraestrutura de orquestração dos serviços criados com Docker. A seção 5.4 detalha a implementação do agente de controle em JaCaMo/Hypermedea. Por fim, será apresentada a validação dos requisitos e a análise dos resultados obtidos.

5.1 MODELAGEM SEMÂNTICA DA BANCADA SEPARATING

Nessa seção, serão apresentados os detalhes da modelagem semântica da bancada Separating. Os tópicos incluem o diagrama do grafo de conhecimento, a definição da estrutura hierárquica das URIs, e a validação semântica. É importante destacar que a modelagem dessa bancada é um exemplo de como a abordagem proposta pode ser aplicada a outras bancadas do laboratório LAI. Esta demonstração evidencia a escalabilidade da abordagem.

5.1.1 Diagrama do Grafo de Conhecimento da Bancada

A representação semântica da planta do LAI foi consolidada em um Grafo de Conhecimento, projetado a partir das ontologias definidas na metodologia. Esta seção detalha a estrutura deste grafo, explicando as decisões de modelagem e as relações entre os componentes da bancada *Separating*. O grafo aqui apresentado servirá como o “plano” para o desenvolvimento do construtor semântico, que integrará a estrutura da aplicação Jena.

Seguindo o princípio de design da modularidade, a representação da linha de montagem FESTO foi estruturada hierarquicamente. O modelo começa com um recurso principal, o sistema MPS Festo (*MPSFesto*), que se subdivide em bancadas, as quais, por sua vez, se subdividem em subsistemas e componentes. Essa abordagem permite que o sistema evolua de forma controlada e consistente, facilitando a adição ou remoção de módulos sem impactar as partes já desenvolvidas do grafo.

O sistema completo foi modelado como uma linha de produção (*ai4i:Production Line*), uma classe específica de domínio que especializa o conceito mais genérico de sistema SSN. Cada bancada individual, por sua vez, foi representada como uma plataforma SOSA (*Platform*). A ontologia SOSA define plataforma como uma subclasse de sistema SSN (*ssn:System*), o que a torna semanticamente válida para ser utilizada como objeto da propriedade *hasSubSystem*, permitindo a construção de uma hierarquia de sistemas aninhados de forma consistente.

A primeira parte do grafo a ser apresentada é justamente a hierarquia que liga

o sistema MPS Festo às bancadas. A Figura 4 ilustra esta relação de alto nível. O recurso principal foi modelado como uma linha de produção AI4Industry, reutilizando o conceito da ontologia do AI4Industry. Para representar o fluxo do processo, as bancadas de início e fim da linha (bancada *Distributing* e bancada *Sorting*) foram conectadas através das propriedades *inputMachine* e *outputMachine*, respectivamente. As demais bancadas intermediárias, incluindo a bancada *Separating*, foram conectadas como partes integrantes da linha de produção através da propriedade *hasSubSystem*, estabelecendo a composição geral da planta.

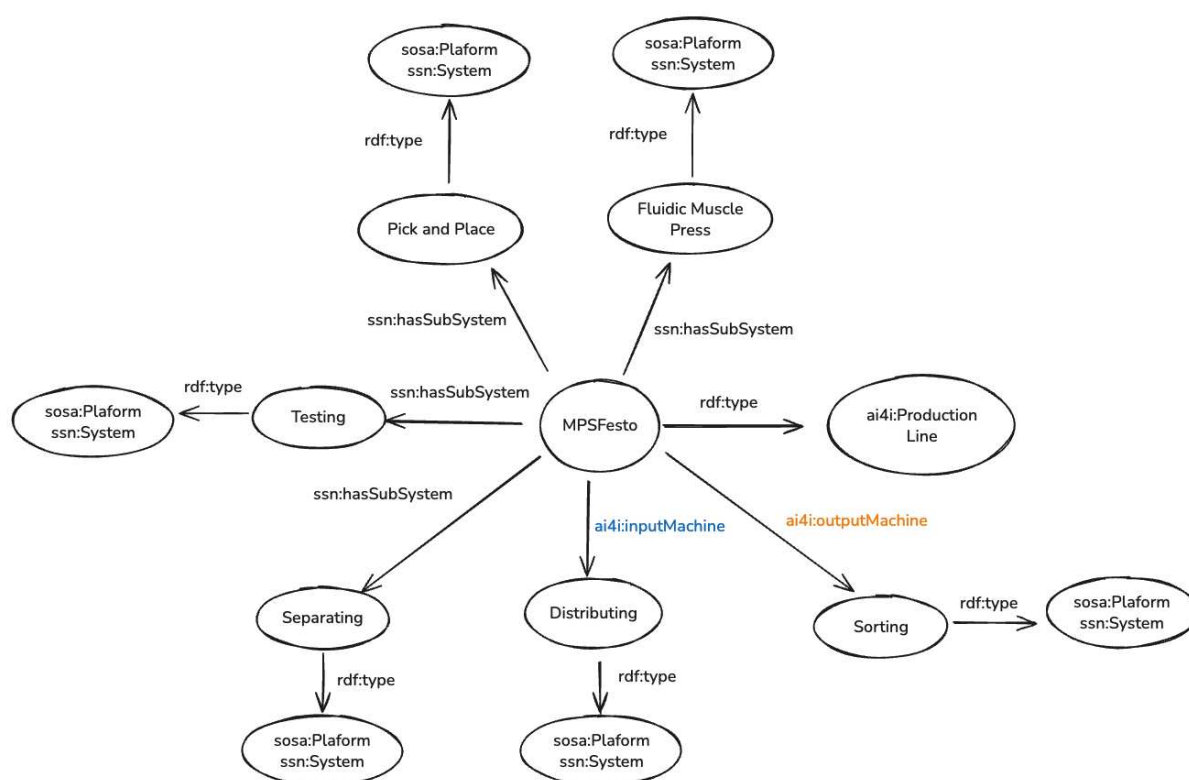


Figura 4 – Relação hierárquica entre a linha de produção MPSFesto e a bancada Separating.

Para modelar a bancada *Separating* em si, foi necessário primeiro analisar sua estrutura física e funcional. O esquemático da bancada, apresentado na Figura 1, ilustra a disposição de seus nove sensores e atuadores principais. A partir desta análise, a decisão de modelagem foi dividir a bancada em três subsistemas lógicos, baseados em suas responsabilidades funcionais, em vez de sua proximidade física.

Esta divisão funcional resultou nos seguintes subsistemas:

- Esteira principal (*MainConveyor*), responsável pelo transporte principal da peça, que abrange a esteira principal (8), o sensor de entrada (1) e o sensor de saída (7);
- Sistema de verificação de orientação (*OrientationVerificationSystem*), que encapsula a lógica de inspeção, que abrange o sensor de orientação (3), o sensor de

peça parada (2) e a trava para inspeção (4);

- Sistema de descarte (*DiscardSystem*), que gerencia o processo de descarte, que abrange o atuador de descarte (6), pela esteira de descarte (9) e pelo sensor de descarte (5).

É importante notar que outras abordagens de divisão seriam possíveis. Por exemplo, uma divisão puramente física poderia considerar que o atuador de descarte (*DiscardDiverter*) é parte da esteira principal (*MainConveyor*), já que está fisicamente posicionado sobre ele. No entanto, a abordagem funcional foi escolhida por ser semanticamente mais robusta, pois isola as capacidades do sistema em módulos coesos, o que facilita a descoberta de funções pelo agente e a manutenção do modelo. As subseções seguintes detalharão a modelagem de cada um destes subsistemas.

5.1.1.1 Modelagem do Subsistema da Esteira Principal

O subsistema da esteira principal (*MainConveyor*) é o responsável pelo fluxo de peças ao longo da bancada. Para representar este e outros dispositivos de transporte de forma consistente, foi definida a classe *Conveyor* como uma especialização de plataforma SOSA. Esta decisão de modelagem permite agrupar todas as esteiras do sistema sob um tipo semântico comum, facilitando consultas e a aplicação de regras genéricas para dispositivos de transporte.

A instância da esteira principal possui duas áreas funcionais, conectadas através das propriedades *inputArea* e *outputArea*. Cada uma dessas áreas (e.g., *EntryPoint*) é modelada como uma característica de interesse SOSA (*sosa:FeatureOfInterest*) que, por sua vez, possui uma propriedade de presença específica (e.g., *EntryPointPresence*) através da relação *ssn:hasProperty*. É esta propriedade que é formalmente ligada ao sensor físico (e.g., *ArrivalSensor*) através da relação *observes*. Esta cadeia de triplas estabelece um caminho de descoberta funcional, permitindo que o agente, partindo da esteira, encontre o sensor responsável por monitorar uma área específica através de uma consulta semântica.

Complementarmente, a relação de contenção física é mantida através da propriedade *sosa:hosts*. A descrição de que a esteira principal hospeda o sensor de chegada descreve que o sensor é um componente físico da plataforma da esteira. A coexistência destes dois caminhos de relacionamento — o funcional e o físico — enriquece o modelo, permitindo que o agente realize diferentes tipos de consulta, seja para descobrir um sensor com base em seu propósito ou para obter todos os componentes de hardware de um subsistema.

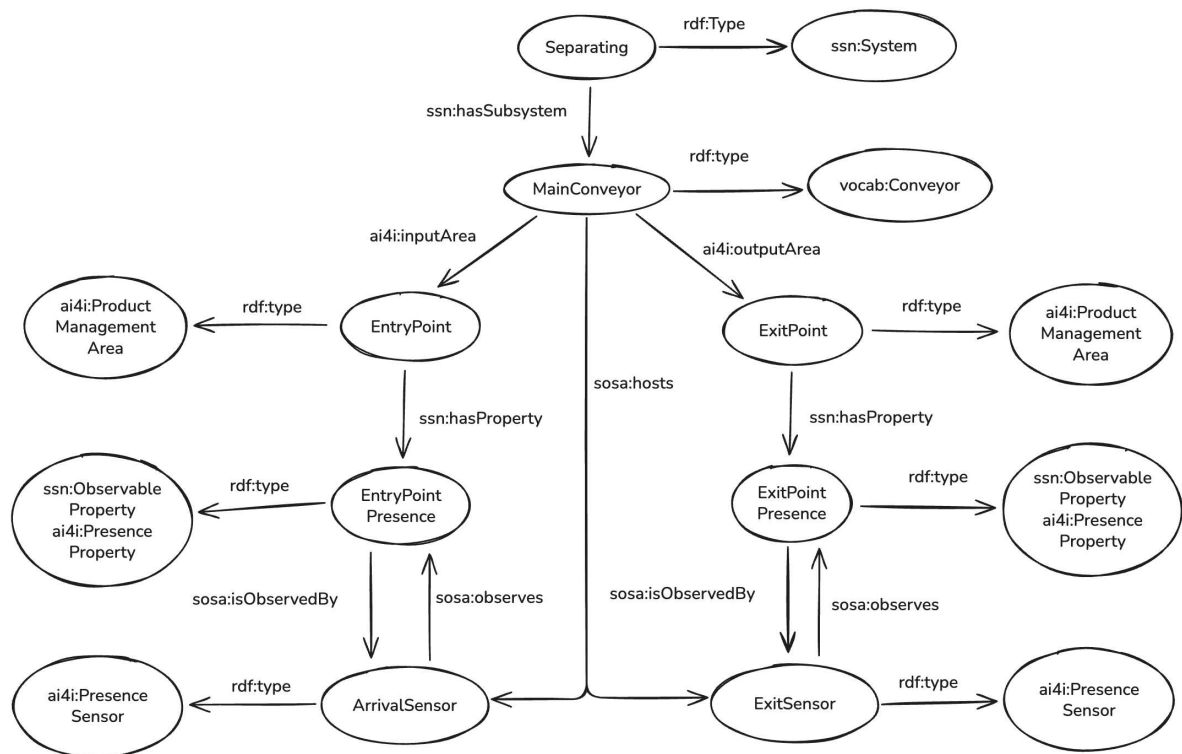


Figura 5 – Grafo de conhecimento do subsistema da esteira principal.

5.1.1.2 Modelagem do Subsistema de Verificação de Orientação

O subsistema de verificação de orientação (*OrientationVerificationSystem*) é o módulo responsável pela inspeção da peça. Modelado como uma plataforma SOSA, ele hospeda os três componentes que executam esta tarefa: o sensor de peça parada (*StoppedSensor*), a trava (*Lock*) e o sensor de orientação (*OrientationSensor*). Para descrever a capacidade do atuador, a ontologia SAREF foi utilizada para criar uma função de travamento (*LockingFunction*). A relação SAREF *accomplishes* estabelece formalmente o propósito do atuador, permitindo que o agente o descubra através de sua funcionalidade de travamento, em vez de seu nome.

A modelagem do sensor de orientação demandou a criação de uma especialização semântica para evitar ambiguidades, foi definida uma nova classe, chamada *MidpointHeight*, como uma subclasse de propriedade observável SOSA. Esta especialização representa formalmente a medição de altura no ponto médio da peça, utilizada para inferir sua orientação. Esta decisão de design permite que o agente realize uma descoberta funcional precisa, consultando o grafo por sensores que observam um objeto da classe *MidpointHeight*, garantindo assim a correta identificação do dispositivo para a tarefa de verificação de orientação.

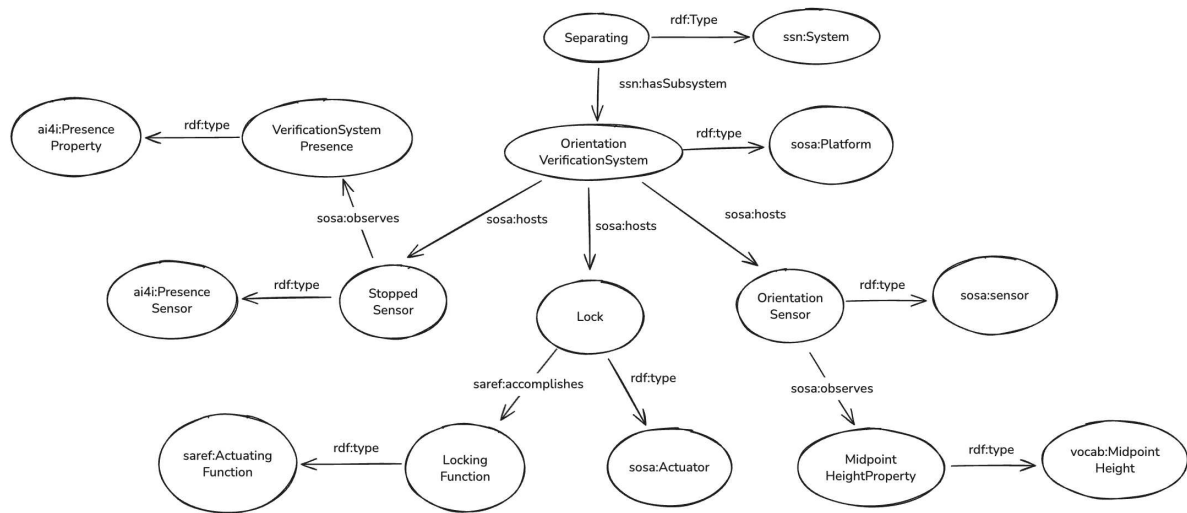


Figura 6 – Grafo de conhecimento do subsistema de verificação de orientação.

5.1.1.3 Modelagem do Subsistema de Descarte

O subsistema de descarte (*DiscardSystem*) encapsula todo o processo de remoção de peças defeituosas. Ele é modelado como uma plataforma SOSA autocontida, cujo fluxo de operação é delimitado por duas áreas funcionais. A relação entre o sistema e essas áreas é estabelecida pelas propriedades da ontologia do *Ai4Industry*. A propriedade *inputArea* conecta o sistema à sua área de entrada (*DiscardInputArea*), que representa o ponto onde o atuador desviador (*DiscardDiverter*) é acionado para iniciar o descarte. De forma análoga, a propriedade *outputArea* liga o sistema à sua área de saída (*DiscardOutputArea*), cujo estado é monitorado pelo sensor de descarte (*DiscardSensor*) para confirmar que a peça foi efetivamente removida. A capacidade do atuador chave, o desviador, é formalmente descrita pela relação SAREF *accomplishes* da função de descarte (*DiscardFunction*).

A conexão entre a área de entrada e o atuador que nela opera é formalizada através do padrão da ontologia SAREF. A área de entrada do sistema possui uma propriedade atuável chamada *DiverterStatusProperty*, conforme a relação *ssn:hasProperty*. A função do atuador (*DiscardFunction*) é então modelada para conter um comando que atua sob esta propriedade específica. Como o atuador *DiscardDiverter* cumpre esta função, cria-se uma ligação semântica indireta, mas formalmente correta e consultável, entre o atuador e a área funcional que ele controla.

5.1.2 Integração do TD com o Grafo de Conhecimento

Enquanto as ontologias SOSA e SAREF descrevem a estrutura hierárquica e o propósito funcional da planta, a ontologia TD é integrada para modelar a camada de interação concreta. A sua função é descrever, de forma padronizada, como um agente pode se comunicar com os recursos do sistema através da rede. Esta seção detalha

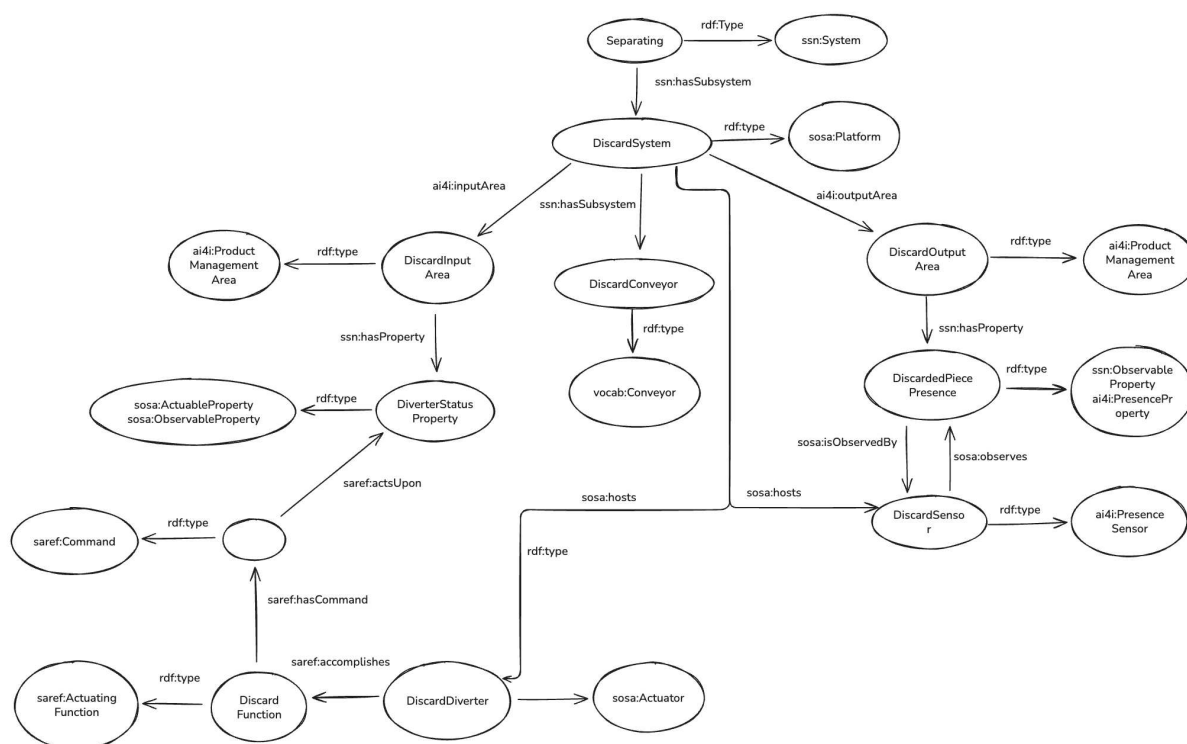


Figura 7 – Grafo de conhecimento do subsistema de descarte.

a decisão de design sobre quais entidades no grafo são consideradas uma *Thing* e como suas *affordances* (propriedades, ações e eventos) são representadas.

A decisão fundamental da arquitetura foi modelar cada sensor e atuador individual como uma instância de *td:Thing*. Esta abordagem de alta granularidade foi escolhida porque cada dispositivo de hardware no sistema corresponde diretamente a um *endpoint* único e endereçável na API REST exposta pelo middleware. A alternativa de modelar um subsistema inteiro, como o subsistema de verificação de orientação, como uma única *Thing* foi deliberadamente descartada. Tal abordagem resultaria em uma TD monolítica, que agruparia as *affordances* de múltiplos dispositivos distintos (o sensor de parada, a trava e o sensor de orientação), tornando ambíguo para o agente qual *affordance* corresponde a qual componente físico. Ao associar as *affordances* diretamente ao dispositivo de nível mais baixo, a arquitetura garante uma ligação um-para-um entre um recurso semântico e sua interface de comunicação.

Esta decisão representa uma mudança arquitetural significativa em relação ao projeto anterior, que motivou este trabalho. No sistema original, a bancada inteira era representada como uma *Thing* monolítica, resultando em uma descrição “plana” que perdia as relações hierárquicas que agora são explicitadas no grafo. A arquitetura atual, por outro lado, implementa um modelo de *Composite Things*. O sistema é um grafo de múltiplas *Things* (os dispositivos), interligadas por relações semânticas e estruturais. O agente autônomo navega nesta estrutura para descobrir um dispositivo específico com base em sua função e, só então, consulta aquele recurso para obter suas *affordances*

de interação.

5.1.3 Princípios de Design e Estrutura da URI

Na representação semântica RDF, as entidades são representadas por URIs, cuja sintaxe genérica é formalmente definida pela IETF na RFC 3986 (Berners-Lee, T.; Fielding; Masinter, 2005). Essas URIs são identificadores únicos e globais que representam os recursos no grafo de conhecimento. Para este projeto, foi estabelecida uma estratégia de nomenclatura consistente, partindo do princípio de que a estrutura da URI deve refletir a hierarquia física e lógica da planta industrial, baseando-se no modelo apresentado na Seção 5.1.1. Esta abordagem aumenta a legibilidade do modelo e fornece um mecanismo de navegação intuitivo tanto para desenvolvedores quanto para o agente consumidor. A estrutura de URI adotada é apresentada no Código 5.1.

```
1 BASE_URI/linha-de-producao/bancada/subsistema/componente/
```

Código 5.1 – Estrutura de URI adotada

Cada segmento do caminho representa um nível na hierarquia de composição do sistema. Por exemplo, o sensor de orientação receberá uma URI como mostrada no Código 5.2, demonstrando que esse sensor faz parte do subsistema de verificação de orientação, e da bancada Separating, seguindo a mesma hierarquia apresentada na Figura 8.

```
1 BASE_URI/MPSFesto/Separating/OrientationSystem/OrientationSensor/
```

Código 5.2 – Exemplo de URI para o sensor de orientação

Esta decisão de design traz três benefícios principais. Primeiro, a legibilidade humana, que facilita a depuração e a manutenção do grafo, pois a URI de um recurso revela imediatamente seu contexto e sua posição na estrutura da planta.

Segundo, ela permite uma navegabilidade lógica. Um agente que está interagindo com um componente pode, através da simples manipulação da string da URI (e.g., removendo o último segmento), descobrir a URI de seu sistema hospedeiro, permitindo-lhe “subir” na hierarquia para obter informações contextuais. Por fim, esta estrutura impede a ambiguidade através da definição de escopo, evitando colisões de nomes entre componentes de diferentes bancadas e estabelecendo um *namespace* claro e organizado para toda a planta do laboratório LAI.

Adicionalmente, esta abordagem de modelagem e nomenclatura reflete diretamente o princípio de design da modularidade. Cada subsistema, como o de verificação de orientação, é modelado como uma entidade autocontida, com suas próprias funções e dispositivos associados. Esta organização modular aumenta a legibilidade e a clareza do modelo, além de facilitar a manutenção e a futura extensão do sistema,

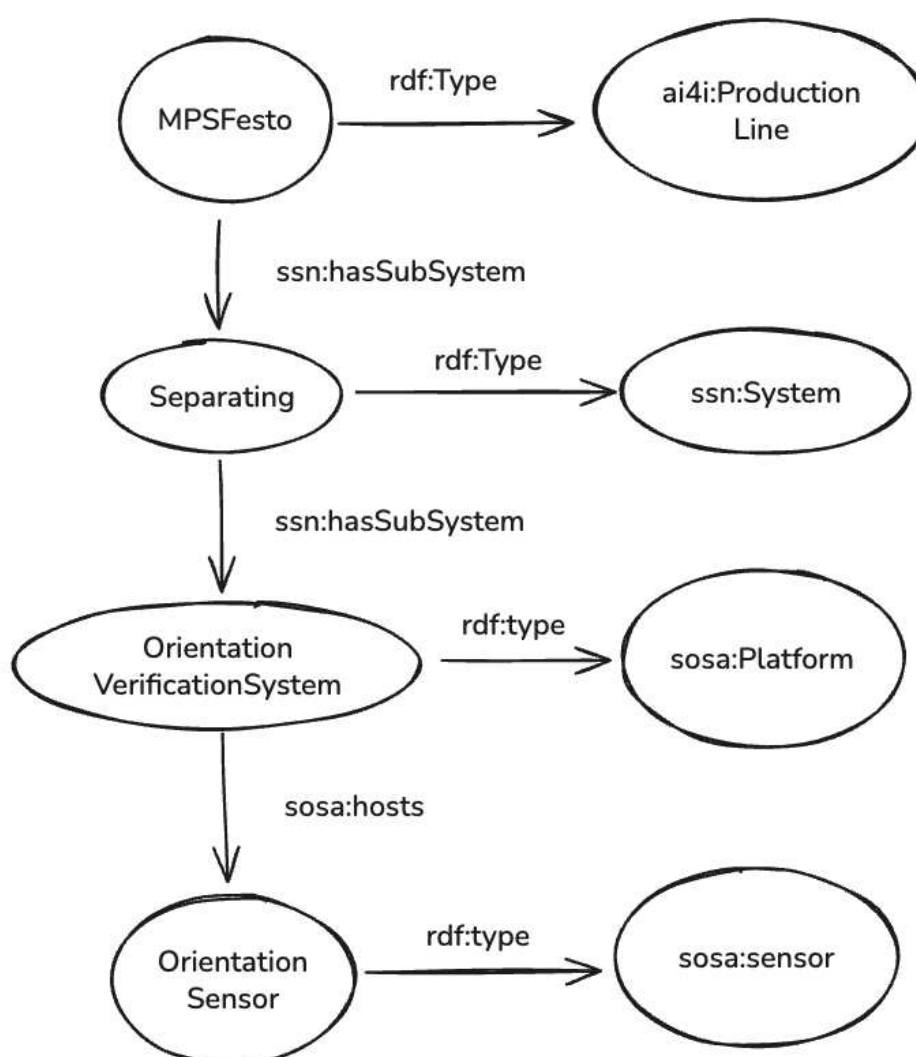


Figura 8 – Grafo de conhecimento resumido do sensor de orientação.

permitindo que novas funcionalidades ou componentes sejam adicionados de forma controlada e consistente.

É importante ressaltar que a URI de base (BASE_URI) é definida de acordo com o ambiente de execução, e é utilizada para todas as URIs do modelo. Sendo assim, em um ambiente de produção, a URI de base seria definida por um domínio de produção, como `https://agents-in-charge.ufsc.br`, mas, para o ambiente de desenvolvimento e testes, a URI de base é definida como `http://localhost/`. Mais detalhes a respeito da infraestrutura de execução do projeto serão apresentados na Seção 5.3.

5.2 IMPLEMENTAÇÃO DA CAMADA DE CONHECIMENTO COM APACHE JENA

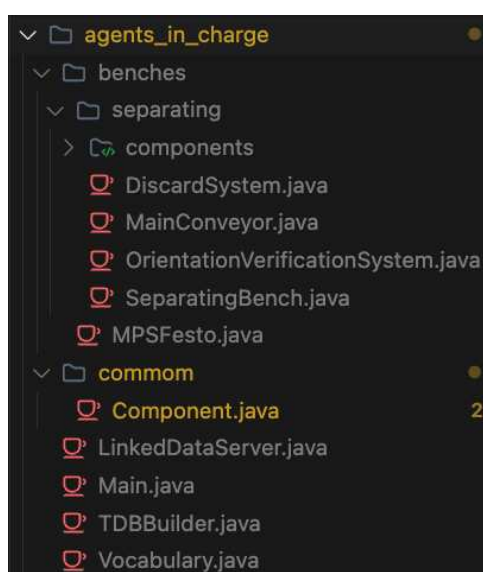
Esta seção abordará a implementação prática do Grafo de Conhecimento, cuja estrutura conceitual foi apresentada no capítulo anterior, utilizando o framework Apache Jena. Serão detalhados os componentes do ecossistema Jena que foram utilizados, a

organização do projeto Java, o gerenciamento das ontologias e, finalmente, as classes e métodos chave que realizam a construção programática do grafo.

5.2.1 Estrutura do Projeto Java

O projeto Java foi estruturado em pacotes que refletem diretamente a hierarquia do modelo semântico. Esta abordagem, ilustrada na Figura 9, visa fazer com que a organização do código seja um espelho da organização lógica do grafo, facilitando a navegação e a futura extensão do projeto.

Figura 9 – Estrutura de pacotes do projeto Java.



Fonte: do autor.

O pacote principal, `br.ufsc.agents_in_charge`, contém as classes de orquestração, como o `TDBBuilder` e o `LinkedDataServer`. Um subpacote `common` abriga interfaces e classes genéricas, como a interface `Component`, que define o contrato para todos os elementos construíveis do grafo. O pacote `benches` contém subpacotes para cada bancada da linha de produção, como `separating`. Dentro deste, encontram-se as classes que modelam os subsistemas da bancada (e.g., `MainConveyor.java`), e um subpacote adicional, `components`, agrupa as classes que representam os dispositivos individuais, como sensores e atuadores.

5.2.2 Importação de Ontologias

Para evitar a repetição de URIs em formato de string ao longo do código, todos os vocabulários e ontologias utilizados foram definidos como classes de constantes em Java. Em vez de criar essas classes manualmente, o que seria um processo propenso a erros, foi utilizada a ferramenta `Schemagen`, parte do ecossistema `Apache Jena`.

O Schemagen é um gerador de código que lê um esquema RDF (como um arquivo de ontologia em Turtle ou RDF/XML) e gera automaticamente uma classe Java que representa as classes e propriedades daquele vocabulário como constantes estáticas.

Este processo foi integrado ao ciclo de vida do projeto através do plugin *jena-maven-tools*. Os arquivos de definição das ontologias padrão (e.g., *sosa.ttl*, *saref.ttl*), obtidos de suas fontes oficiais, foram adicionados ao diretório *src/main/resources/vocabs/* do projeto. A configuração do plugin no arquivo *pom.xml*, detalhada no Código 5.3, foi então ajustada para instruir o Maven, gerenciador de dependências, a, durante a fase de compilação, executar o Schemagen sobre cada um desses arquivos, gerando as classes Java correspondentes (e.g., *Sosa.java*, *Saref.java*) diretamente no pacote *br.ufsc.agents_in_charge*.

```
1 <plugin>
2   <groupId>org.apache.jena</groupId>
3   <artifactId>jena-maven-tools</artifactId>
4   <version>3.6.0</version>
5   <configuration>
6     <includes>
7       <include>src/main/resources/vocabs/*.ttl</include>
8       <include>src/main/resources/vocabs/*.xml</include>
9     </includes>
10    <fileOptions>
11      <source>
12        <input>default</input>
13        <package-name>br.ufsc.agents_in_charge</package-name>
14      </source>
15      <source>
16        <input>src/main/resources/vocabs/ssn.ttl</input>
17        <ontology>true</ontology>
18      </source>
19      <!-- Outras ontologias -->
20    </fileOptions>
21  </configuration>
22  <executions>
23    <execution>
24      <id>schemagen</id>
25      <goals>
26        <goal>translate</goal>
27      </goals>
28    </execution>
29  </executions>
30 </plugin>
```

Código 5.3 – Configuração do plugin Schemagen no arquivo *pom.xml*.

O resultado deste processo automatizado é uma classe robusta para cada voca-

bulário, como o trecho da classe `Sosa.java` gerada, exemplificado no Código 5.5. Ao utilizar estas constantes geradas no código de construção do grafo, como mostrado no Código 5.4, garante-se a uniformidade e a correção das URIs, elimina-se a possibilidade de erros de digitação e centraliza-se a gestão dos vocabulários de forma segura e de fácil manutenção.

```

1 Resource separatingBench = model.createResource(BASE_URI +
  "SeparatingBench");
2 model.add(separatingBench, RDF.type, Sosa.Platform);

```

Código 5.4 – Exemplo de aplicação da classe `Sosa.java`

```

1  /* CVS $Id: $ */
2  package br.ufsc.agents_in_charge;
3  import org.apache.jena.rdf.model.*;
4  import org.apache.jena.ontology.*;
5
6  /**
7   * Vocabulary definitions from file:/src/main/resources/vocabs/sosa.ttl
8   * @author Auto-generated by schemagen on 28 Jun 2025 22:08
9   */
10 public class Sosa {
11     /** <p>The ontology model that holds the vocabulary terms</p> */
12     private static final OntModel M_MODEL =
13         ModelFactory.createOntologyModel( OntModelSpec.OWL_MEM, null );
14
15     /** <p>The namespace of the vocabulary as a string</p> */
16     public static final String NS = "http://www.w3.org/ns/sosa/";
17
18     /** <p>The namespace of the vocabulary as a string</p>
19      * @return namespace as String
20      * @see #NS */
21     public static String getURI() {return NS;}
22
23     /** <p>The namespace of the vocabulary as a resource</p> */
24     public static final Resource NAMESPACE = M_MODEL.createResource(
25         NS );
26
27     /** <p>A Platform is an entity that hosts other entities,
28      * particularly Sensors, Actuators,
29      * Samplers, and other Platforms.</p>
30      */
31     public static final OntClass Platform = M_MODEL.createClass(
32         "http://www.w3.org/ns/sosa/Platform" );
33 }

```

Código 5.5 – Trecho da classe `Sosa.java` gerada automaticamente pelo Schemagen.

5.2.3 Construção Programática do Grafo

Conforme a arquitetura definida na Figura 3, a construção do Grafo de Conhecimento é realizada de forma programática, separadamente do domínio de exposição do grafo. Esta abordagem foi escolhida visando a consistência, a manutenibilidade e a modularidade do modelo, em contraste com a edição manual de arquivos RDF, que é propensa a erros. Esta seção detalha as principais classes e padrões de projeto utilizados na implementação do construtor do grafo em Java.

5.2.3.1 A Classe *Vocabulary*

A base para a construção consistente do grafo é a classe *Vocabulary*. Sua principal responsabilidade é centralizar a definição de todas as constantes de URI utilizadas no projeto. Isso inclui as URIs base para os recursos do grafo (`BASE_URI`), para o vocabulário customizado (`VOCAB_URI`) e para o servidor WoT (`SERVER_URI`), que aponta para a API exposta pelo middleware Node-RED. Além disso, a classe contém um método estático, `build(Model model)`, que é responsável por popular o grafo com as definições do vocabulário customizado, como as classes `vocab:Conveyor` e `vocab:MidpointHeight`, garantindo que estas estejam disponíveis antes da instanciamento dos componentes da planta.

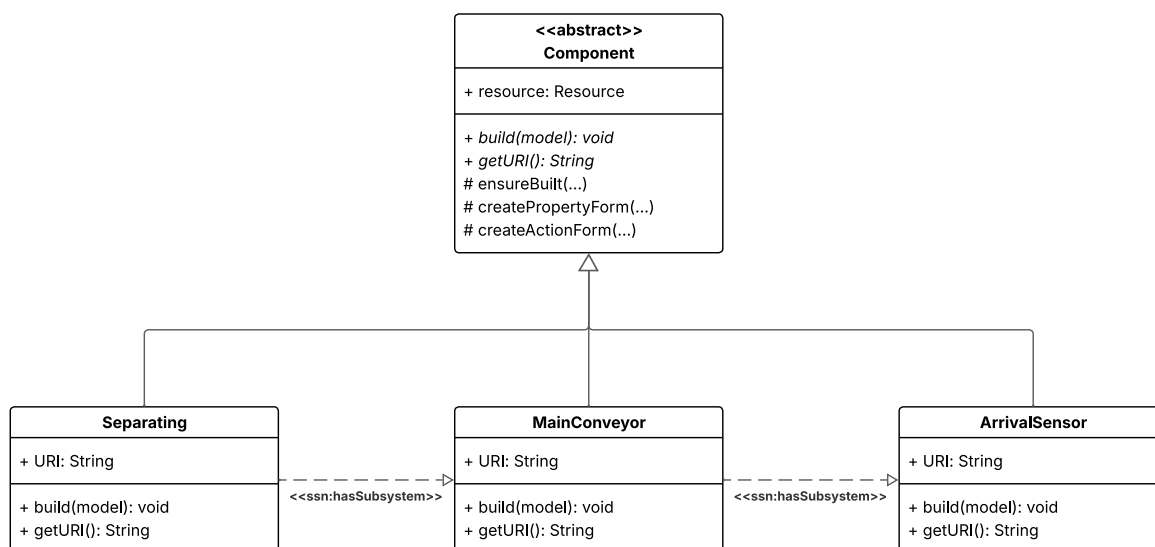
5.2.3.2 Arquitetura de Classes para a Construção do Grafo

Objetivando uma construção modular e hierárquica do grafo, foi projetada uma arquitetura de classes em Java, ilustrada na Figura 10. A base desta arquitetura é a classe abstrata *Component*, que estabelece uma interface comum para todos os elementos que serão representados no grafo, desde a linha de produção completa (*MPSFesto*) até os sensores e atuadores (e.g., *ArrivalSensor*).

Como demonstrado no diagrama, todas as classes que representam um recurso (e.g., *SeparatingBench*, *MainConveyor*, *ArrivalSensor*) herdam da classe *Component*. Elas devem implementar o método abstrato `build(Model model)`, que contém a lógica específica para adicionar as triplas daquele componente ao modelo RDF. Esta abordagem estabelece uma interface padronizada que todos os componentes devem seguir, facilitando a criação consistente do grafo RDF e permitindo uma estrutura hierárquica natural.

Além da relação de herança entre os componentes, o diagrama também utiliza setas tracejadas com estereótipos (e.g., «`ssn:hasSubsystem`») para representar relações semânticas entre os componentes modelados. Essas relações não correspondem a atributos ou associações estruturais no código-fonte, mas sim a ligações estabelecidas dinamicamente durante a execução do método `build`.

Figura 10 – Diagrama de classes da arquitetura de construção do grafo.



Fonte: do autor.

A classe `Component` também inclui métodos utilitários para promover a reutilização de código. Os métodos `createPropertyForm` e `createActionForm` padronizam a criação das *affordances* da TD, garantindo que todas as interfaces de interação sigam a mesma estrutura. O método `ensureBuilt`, por sua vez, é crucial para a construção hierárquica: antes de um componente complexo, como uma bancada, adicionar um link para um de seus subsistemas, ele invoca `ensureBuilt` para garantir que o subsistema já tenha sido construído no grafo, evitando referências a recursos vazios e garantindo a ordem correta de construção.

5.2.3.3 Exemplos de Implementação de Componentes

A aplicação do padrão `Component` pode ser observada na prática nas classes que representam os elementos da planta. A classe `SeparatingBench`, mostrada no Código 5.6, por exemplo, implementa seu método `build` orquestrando a construção de seus subsistemas. Ela primeiro invoca `ensureBuilt` para cada um de seus subsistemas (`MainConveyor`, `OrientationVerificationSystem` e `DiscardSystem`) e, em seguida, adiciona as triplas que a conectam a eles através da propriedade `ssn:hasSubSystem`.

```

1 public class SeparatingBench extends Component {
2     public static final String URI = MPSFesto.URI + "Separating/";
3
4     public SeparatingBench() {
5         super();
6     }
7

```

```

8      @Override
9      public String getURI() {
10         return URI;
11     }
12
13     @Override
14     public void build(Model model) {
15         this.resource = model.createResource(URI);
16         resource.addProperty(RDF.type, Sosa.Platform);
17
18         ensureBuilt(model, OrientationVerificationSystem.class);
19         ensureBuilt(model, DiscardSystem.class);
20         ensureBuilt(model, MainConveyor.class);
21
22         // Adiciona as relacoes de composicao
23         resource.addProperty(Ssn.hasSubSystem,
24             model.getResource(OrientationVerificationSystem.URI));
25         // ...
26     }
27 }

```

Código 5.6 – Implementação do método build na classe SeparatingBench.

No nível mais baixo da hierarquia, uma classe como DiscardDiverter, mostrada no Código 5.7, implementa seu método build para adicionar todas as suas propriedades semânticas e de interação, como seu tipo (sosa:Actuator e td:Thing), a função que realiza (saref:accomplishes) e suas *affordances* de ação e propriedade, utilizando os métodos utilitários herdados de Component.

```

1  @Override
2  public void build(Model model) {
3      ensureBuilt(model, DiverterStatusProperty.class);
4      ensureBuilt(model, DiscardFunction.class);
5
6      this.resource = model.createResource(URI);
7      resource.addProperty(RDF.type, Td.Thing);
8      resource.addProperty(RDF.type, Sosa.Actuator);
9      resource.addProperty(Saref.accomplishes,
10         model.getResource(DiscardFunction.URI));
11     resource.addProperty(RDFS.label, "Discard Diverter");
12     resource.addProperty(RDFS.comment,
13         model.createLiteral("Desviador de peças para o descarte",
14             "pt-BR"));
13     resource.addProperty(Saref.accomplishes,
14         model.getResource(DiscardFunction.URI));
15
16     // Property affordance for reading discard diverter status
17     // ... Implementação ...

```

```
17 // Action affordance for activating discard diverter
18 Resource actionAffordance = model.createResource();
19 actionAffordance.addProperty(RDF.type, Td.ActionAffordance);
20 actionAffordance.addProperty(RDFS.label, "Discard diverter for part");
21 actionAffordance.addProperty(Td.hasForm, createActionForm(model,
22 Vocabulary.SERVER_URI + "discardDiverter"));
23
24 resource.addProperty(Td.hasActionAffordance, actionAffordance);
25 }
```

Código 5.7 – Implementação do método build na classe DiscardDiverter.

5.2.3.4 O Orquestrador TDBBuilder

Finalmente, a classe TDBBuilder atua como o orquestrador de todo o processo. Sua responsabilidade é inicializar a conexão com o banco de dados TDB2, obter o objeto Model e iniciar o processo de construção em cascata. Como pode ser visto no Código 5.8, o método buildTDB primeiro constrói as definições do vocabulário e, em seguida, instancia e constrói apenas o componente de mais alto nível da hierarquia, o :MPSFesto. Devido ao uso do método ensureBuilt, a chamada MPSFesto.build(model) desencadeia a construção de todos os seus subsistemas e componentes de forma recursiva e na ordem correta, resultando no grafo de conhecimento completo e persistido.

```
1 public void buildTDB() {
2     // ... inicialização do dataset e transação ...
3     try {
4         Model model = dataset.getDefaultModel();
5         // Constroi as definicoes do vocabulario customizado
6         Vocabulary.build(model);
7
8         // Inicia a construcao em cascata a partir do componente raiz
9         Component mpsFesto = new MPSFesto();
10        mpsFesto.build(model);
11
12        dataset.commit();
13    } finally {
14        // ... finalizacao da transacao ...
15    }
16 }
```

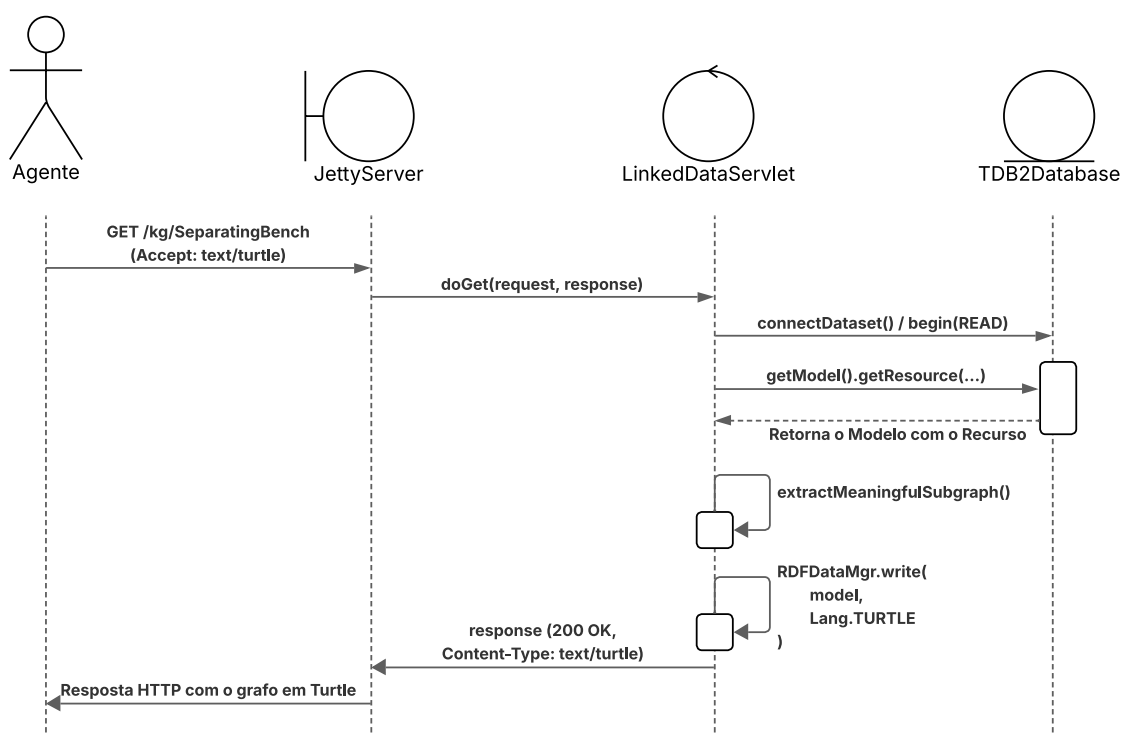
Código 5.8 – Método principal da classe TDBBuilder.

5.2.4 Serialização e Exposição do Grafo Gerado

Após a construção e persistência do Grafo de Conhecimento no banco de dados TDB2, a etapa final da camada de conhecimento é sua exposição para o agente consumidor. Para atender ao requisito de Descoberta Semântica (RF03), foi implementado um servidor *Linked Data*, que expõe os recursos do grafo via HTTP e suporta a negociação de conteúdo para diferentes formatos de serialização RDF.

A implementação do servidor foi realizada em Java, utilizando o servidor web embarcado *Jetty*. A lógica de tratamento das requisições está encapsulada na classe *LinkedDataServlet*, que atua como um intermediário entre as requisições HTTP do agente e o banco de dados TDB2. O fluxo de uma requisição para um recurso específico, como a bancada *Separating*, é ilustrado no diagrama de sequência da Figura 11.

Figura 11 – Diagrama de sequência para uma requisição ao Servidor *Linked Data*.



Fonte: do autor.

O processo, conforme detalhado no diagrama, ocorre da seguinte forma:

1. O Agente Consumidor (e.g., o agente JaCaMo/Hypermedea) envia uma requisição HTTP GET para a URI de um recurso, como /kg/MPSFesto/Separating. O cabeçalho Accept da requisição indica o formato de serialização desejado (e.g., text/turtle).

2. O servidor Jetty recebe a requisição e a encaminha para o `LinkedDataServlet`.
3. O `Servlet` analisa a URI da requisição, extrai o identificador do recurso e abre uma transação de leitura com o banco de dados TDB2.
4. Uma consulta é feita ao TDB2 para extrair o subgrafo relevante para o recurso solicitado. O método `extractMeaningfulSubgraph` é invocado para garantir que todas as propriedades do recurso, incluindo as descrições de nós em branco (*Blank Nodes*, como as *affordances*), sejam incluídas no resultado.
5. Com o subgrafo em memória, o `Servlet` utiliza o componente `RDFDataMgr` do Apache Jena para serializar o modelo. Com base no cabeçalho `Accept` da requisição original, o Jena seleciona o serializador correto (Turtle, JSON-LD, RDF/XML).
6. Finalmente, o grafo serializado é escrito no corpo da resposta HTTP, que é enviada de volta ao agente.

Esta arquitetura garante que o requisito de exposição do grafo (RF02) seja atendido. A utilização dos serializadores nativos do Jena simplifica a implementação da negociação de conteúdo e garante a geração de documentos RDF válidos e bem formatados. A implementação completa da classe `LinkedDataServer` pode ser consultada no Apêndice A.

5.3 ORQUESTRAÇÃO DE SERVIÇOS COM DOCKER

Para garantir a portabilidade e a consistência do ambiente de desenvolvimento, toda a arquitetura de software foi containerizada utilizando a plataforma Docker (Docker, Inc., 2025). A orquestração dos múltiplos serviços que compõem a solução é gerenciada através de um único arquivo de configuração, o `docker-compose.yml`, que permite iniciar todo o sistema com um único comando.

5.3.1 Arquitetura de Contêineres e o Docker Compose

A arquitetura da solução é composta por um conjunto de microsserviços, onde cada um é executado em seu próprio contêiner isolado. Esta abordagem modular simplifica o desenvolvimento e o deploy de cada parte do sistema. Os principais serviços definidos no arquivo `docker-compose.yml` são:

- **Nginx:** Atua como um proxy reverso, servindo como o ponto de entrada único para a aplicação e direcionando as requisições HTTP para os serviços internos.
- **Dashboard:** Uma aplicação web que fornece uma interface gráfica para interação com o sistema, visualização do estado dos sensores e adição de peças no ambiente simulado.

- **WoT-Server:** Uma instância do Node-RED que implementa o middleware WoT, comunicando-se com a camada física via OPC UA e expondo a API REST.
- **Semantic-App:** O servidor Linked Data, implementado em Java, responsável por expor o Grafo de Conhecimento para o agente.
- **OPC-Simulator:** O simulador da planta, que expõe um servidor OPC UA na porta 4840, replicando a interface do CLP real.
- **Bob-Agent:** O agente de controle, implementado utilizando a plataforma JaCaMo, que utiliza o Grafo de Conhecimento para operar a bancada de forma semântica.

Adicionalmente, um serviço semantic-builder foi configurado para executar a construção do banco de dados do Grafo de Conhecimento sob demanda.

5.3.2 Replicação e Perfis de Execução

A principal vantagem da containerização é a facilidade de replicação do ambiente. Com o Docker e o Docker Compose, um desenvolvedor pode iniciar todos os serviços necessários com o comando mostrado no Código 5.10, eliminando a necessidade de configurar manualmente cada dependência e garantindo um ambiente de desenvolvimento consistente entre diferentes máquinas.

Para gerenciar a transição entre os ambientes de desenvolvimento e produção, foi utilizado o recurso de perfis (profiles) do Docker Compose. O serviço `opc-simulator` foi associado ao perfil `dev`. Para executar o ambiente de desenvolvimento, utiliza-se o comando mostrado no Código 5.9, que inclui o simulador. Para operar no ambiente de produção, o comando mostrado no Código 5.10 é executado sem o perfil, o que inicia todos os serviços exceto o simulador. Nesta configuração, o serviço `wot-server` é então configurado para se conectar ao endereço de IP do CLP real. Esta estratégia simplifica o processo de deploy e a alternância entre os ambientes de teste e produção.

```
1 docker-compose --profile dev up
```

Código 5.9 – Comando para iniciar o ambiente de desenvolvimento.

```
1 docker-compose up
```

Código 5.10 – Comando para iniciar o ambiente de produção.

5.4 IMPLEMENTAÇÃO E OPERAÇÃO DO AGENTE DE CONTROLE

Para validar a eficácia da camada de conhecimento e demonstrar na prática o desacoplamento proposto, foi desenvolvido um agente de software. Esta seção descreve a arquitetura deste agente, sua implementação e, crucialmente, como ele utiliza o Grafo de Conhecimento para operar a bancada de forma semântica.

5.4.1 Estrutura do Agente BDI em JaCaMo/Hypermedea

O agente de controle foi implementado utilizando a plataforma JaCaMo, que integra o modelo de raciocínio BDI (através da linguagem *Jason*) com um ambiente para artefatos (CArtAgO). A interação com o Grafo de Conhecimento foi viabilizada pelo framework Hypermedea, que se integra ao CArtAgO e permite que o agente perceba e navegue em ambientes *Linked Data*.

A lógica do agente foi estruturada em um conjunto de planos hierárquicos, projetados para serem orientados a objetivos de alto nível, como `!processWorkpiece`. A descoberta dos componentes é realizada de forma semântica: o agente busca por um componente com base em suas classes ontológicas e nas relações que ele possui, não em seu identificador. A base de conhecimento do agente é composta por um conjunto de regras lógicas que mapeiam os predicados RDF (e.g., `ssn:hasSubSystem`, `saref:accomplishes`) para crenças internas, permitindo que ele raciocine sobre a estrutura e as funções descritas no grafo.

5.4.2 O Ciclo de Descoberta Semântica em Ação

O processo operacional do agente inicia-se com o Hypermedea realizando uma requisição HTTP para a URI raiz do Grafo de Conhecimento (`/kg/`). A partir deste ponto de entrada, o framework realiza um processo de *crawling*, navegando recursivamente pelas relações do grafo para popular a base de crenças do agente com as triplas que descrevem a planta.

A principal diferença entre a abordagem semântica e uma implementação direta reside na forma como o agente acessa os recursos. Uma abordagem que acessa os componentes por suas URIs estáticas, como `!readProperty("../ArrivalSensor/", ...)`, embora funcional, recria o problema do acoplamento nominal, tornando o agente frágil a qualquer mudança na URI dos dispositivos.

Em contraste, o agente desenvolvido para este trabalho opera através de um ciclo de descoberta semântica. Para cumprir um objetivo, ele executa uma sequência de consultas lógicas sobre suas crenças para encontrar o recurso apropriado. Um exemplo representativo é a descoberta do atuador de descarte. O plano `!findDiscardDiverter`, que recebe como parâmetro a área de entrada do sistema de descarte (já descoberta semanticamente), segue a seguinte cadeia de inferência para identificar o atuador correto:

1. Descobre qual propriedade está associada à área de entrada.
2. Encontra o comando que atua sobre essa propriedade.
3. Identifica a função que possui esse comando.
4. Finalmente, encontra o atuador que realiza essa função.

Apenas após esta descoberta o agente obtém a URI do atuador para então buscar sua *affordance* de interação. O Código 5.11 demonstra esse plano de descoberta implementado no agente.

```

1 +!findDiscardDiverter(InputArea, DiscardDiverter) <-
2   ?hasProperty(InputArea, DiverterStatus)
3   & actsUpon(Command, DiverterStatus)
4   & hasCommand(DiscardFunction, Command)
5   & accomplishes(DiscardDiverter, DiscardFunction) ;
6 .

```

Código 5.11 – Exemplo de plano de descoberta semântica no agente Jason.

Uma vez que um dispositivo é descoberto, o agente utiliza planos genéricos, como `!readProperty` e `!invokeAction`, que recebem a URI do dispositivo como parâmetro. Estes planos são responsáveis por extrair a *affordance* correspondente do grafo, obter a URL do *endpoint* e realizar a requisição HTTP. Esta arquitetura de agente, que separa a lógica de descoberta da lógica de interação, valida o desacoplamento proposto e demonstra a viabilidade de um controle autônomo e robusto a mudanças na infraestrutura física.

A orquestração geral da lógica da bancada é gerenciada pelos planos de alto nível. O plano `!startProcessingFlow` inicia o ciclo aguardando a chegada de uma peça na área de entrada, através da descoberta do sensor correspondente. Uma vez que a peça é detectada, ele invoca o plano `!processWorkpieceInSystem`, que é o coração do processo. Este plano ativa a esteira principal e, em seguida, descobre e invoca o subsistema de verificação de orientação. Com base no resultado da verificação, o plano decide dinamicamente se deve acionar o subsistema de descarte ou permitir que a peça continue até a área de saída, demonstrando a capacidade do agente de executar um fluxo de trabalho complexo baseado em percepções e descobertas semânticas. Ambos os planos são apresentados no Código 5.12.

```

1 +!startProcessingFlow(System) <-
2   .print("Iniciando fluxo de processamento...");
3   !findConveyorInSystem(System, MainConveyor);
4   ?inputArea(MainConveyor, InputArea);
5   !waitForWorkpieceArrival(InputArea);
6   !processWorkpieceInSystem(System, MainConveyor);
7   .print("Fluxo concluído!");
8   !startProcessingFlow(System);
9 .
10
11 +!processWorkpieceInSystem(System, Conveyor) <-
12   .print("Processando peça no sistema: ", System);
13   !invokeAction(Conveyor, true);
14
15   // Verifica se tem sistema de verificação de orientação

```

```

16  !findOrientationSystem(System, OrientationSystem);
17  !performOrientationCheck(OrientationSystem, Result); // Verifica se a
    peça está na posição correta
18
19  // Executa ação de descarte, se necessário
20  if (Result == true) {
21      !discardWorkpiece(System, Result);
22  } else {
23      ?outputArea(Conveyor, OutputArea);
24      !waitForWorkpieceExit(OutputArea);
25  }
26  !invokeAction(Conveyor, false);
27  .

```

Código 5.12 – Orquestração geral da lógica da bancada no agente Jason.

5.5 VERIFICAÇÃO DE REQUISITOS E ANÁLISE DE RESULTADOS

Este capítulo apresenta a verificação, validação e análise crítica da solução proposta, com o objetivo de demonstrar sua eficácia na resolução do problema de acoplamento semântico. A análise inicia-se com uma verificação sistemática do atendimento aos requisitos funcionais e não-funcionais definidos no Capítulo 3. Em seguida, a hipótese central do trabalho — a redução do acoplamento — é validada empiricamente através da execução de um cenário de teste que simula uma alteração no hardware da planta. Com a funcionalidade e o desacoplamento comprovados, a seção prossegue para uma discussão comparativa, contrastando as vantagens e limitações da nova arquitetura semântica com a abordagem anterior.

5.5.1 Verificação do Atendimento aos Requisitos

Esta seção tem como objetivo verificar se a solução implementada atende aos requisitos funcionais (RF) e não-funcionais (RNF) definidos na Seção 3.5. O desenvolvimento foi realizado em conformidade com todas as restrições de projeto (RP) estabelecidas, como a utilização de vocabulários padronizados e a compatibilidade com a infraestrutura de middleware existente.

O requisito de **Modelagem Semântica da Planta (RF01)** foi atendido através da construção do Grafo de Conhecimento, detalhada na Seção 5.2. O modelo implementa uma estrutura hierárquica, partindo da linha de produção :MPSFesto, que contém a bancada :SeparatingBench como um subsistema. Esta, por sua vez, é decomposta em seus próprios subsistemas lógicos, como o :OrientationVerificationSystem, que abriga os dispositivos individuais. As capacidades de cada componente foram representadas utilizando as ontologias SOSA e SAREF, conforme definido na metodologia.

Quanto à **Exposição do Grafo de Conhecimento (RF02)**, o desenvolvimento do servidor Linked Data, descrito na Subseção 5.2.4, atende a este requisito. A aplicação, implementada com Jetty e Apache Jena, expõe o grafo através de um *endpoint* HTTP. A utilização dos serializadores nativos do Jena garante o suporte à negociação de conteúdo, permitindo que o grafo seja consumido em múltiplos formatos, como Turtle e JSON-LD.

Além do atendimento ao RF02, o servidor *Linked Data* atende também o requisito de **Descoberta Semântica de Capacidades (RF03)**. Ele fornece um ponto de entrada (*/kg/*) a partir do qual o agente pode-se observar todos os recursos reunidos em um único documento RDF. Ademais, é possível acessar cada objeto RDF individualmente, através de suas URIs. Iniciando na URI que representa a linha de produção completa (*/kg/MPSFesto*) é possível navegar para os subsistemas, como a bancada *Separating*, que por sua vez contém os seus próprios subsistemas e componentes.

O atendimento ao requisito de **Execução de Controle Baseada em Semântica (RF04)** foi atendido através das relações entre os componentes do grafo e as *affordances* da TD, que permitem que o agente, após navegar pelo grafo, extraia as informações de interação para executar a operação de controle desejada.

No que tange aos requisitos não-funcionais, a **Escalabilidade (RNF01)** da solução é evidenciada pela arquitetura modular do grafo. Essa arquitetura foi projetada para permitir a adição de uma nova bancada, que pode ser feita através da inclusão de um novo subsistema ao grafo, também pertencente ao sistema *MPSFesto*, como foi feito para a bancada *Separating*.

Por fim, a **Manutenibilidade (RNF02)** foi um dos principais focos do projeto. Ela é atendida por múltiplos fatores, como a construção programática do grafo com Java, a organização modular do código-fonte e a containerização com Docker, que simplifica a configuração do ambiente. A validação empírica deste requisito, que comprova a capacidade de alterar o hardware sem modificar o agente, será o foco da análise na seção seguinte.

5.5.2 Validação do Desacoplamento Semântico

Para validar empiricamente que a arquitetura semântica reduz o acoplamento entre o agente e o hardware, foi executado um cenário de teste de adaptabilidade. Este cenário foi projetado para simular uma alteração comum em um ambiente industrial: a substituição de um componente de hardware, que poderia resultar na alteração de sua interface de comunicação, além de sua URI no grafo de conhecimento.

O procedimento de validação foi dividido em duas etapas. Na primeira, o sistema completo foi executado com o Grafo de Conhecimento original, e o comportamento do agente foi registrado para servir como referência. Na segunda etapa, o Grafo de Conhecimento foi modificado para simular a substituição do desviador de descarte por

um atuador que modifica a direção de esteira principal. Com essa alteração a URI de seu *endpoint* no *middleware* WoT foi alterada, além de sua descrição semântica, garantindo que haja alinhamento entre o grafo de conhecimento e o ambiente físico. O agente foi então executado novamente, sem qualquer alteração em seu código-fonte. O critério de sucesso é a capacidade do agente de operar a bancada corretamente em ambas as etapas, demonstrando sua adaptação à mudança.

O grafo RDF, gerado pelo Jena em Turtle, que representa o novo componente denominado *Track Switch* é apresentado no Código 5.13.

```

1 <http://localhost/kg/MPSFesto/Separating/DiscardSystem/TrackSwitch/>
2   a                               sosa:Actuator , td:Thing;
3   rdfs:comment                     "Switch para alterar a direção da esteira
   para o descarte"@pt-BR;
4   rdfs:label                       "Track Switch"; # Altera a label e comentá
   rio
5   saref:accomplishes               <http://localhost/kg/MPSFesto/Separating/
   DiscardSystem/DiscardFunction/>; # Mantém a mesma função
6   td:hasActionAffordance [ ...
7     td:hasForm [ ...
8       hctl:hasTarget               "http://localhost/api/
   separating/trackSwitch" # Altera também o endpoint
   para acesso
9     ]
10  ];

```

Código 5.13 – Fragmento da alteração do component Track Switch.

5.5.2.1 Execução com o Grafo de Conhecimento Original

Os resultados de cada etapa são apresentados através de uma tabela de eventos extraída dos *logs* do simulador. É importante notar que estes *logs* representam a "verdade" do ambiente físico virtual e não a base de crenças do agente. A informação de que uma peça é "alta", por exemplo, existe no simulador desde sua criação, mas o agente só adquire este conhecimento após realizar a leitura do sensor de orientação.

Na primeira etapa, o agente foi instruído a processar duas peças: uma com altura incorreta (peça 1) e outra com altura correta (peça 2). A Tabela 3 resume a sequência de eventos principais registrados pelo simulador.

A análise da Tabela 3 comprova que o agente operou a bancada conforme o esperado. A peça 1, alta, foi corretamente desviada para o descarte, enquanto a peça 2, com altura correta, prosseguiu até o final da esteira. Este resultado serve como a linha de base para a próxima etapa.

Tabela 3 – Sequência de eventos principais da execução com o grafo original.

Componente	Evento	Observação
— <i>Processamento da Peça 1 (Invertida)</i> —		
Sistema	Peça 1 criada	Atributo 'isHigh = true'
Arrival Sensor	Detecta peça	Estado muda para ON
Main Conveyor	Ativado	Agente inicia transporte
Stopped Sensor	Detecta peça	Peça na posição de inspeção
Orientation Sensor	Detecta peça alta	Estado muda para ON
Discard Diverter	Ativado	Agente inicia o descarte
Discard Conveyor	Ativado	Transporte para o descarte
Sistema	Peça 1 descartada	Fim do ciclo da peça 1
— <i>Processamento da Peça 2 (Normal)</i> —		
Sistema	Peça 2 criada	Atributo 'isHigh = false'
Arrival Sensor	Detecta peça	Estado muda para ON
Main Conveyor	Ativado	Agente inicia transporte
Stopped Sensor	Detecta peça	Peça na posição de inspeção
Orientation Sensor	Não detecta peça alta	Estado permanece OFF
Lock	Desativado	Agente libera a peça
Exit Sensor	Detecta peça	Peça na saída da bancada
Sistema	Peça 2 concluída	Fim do ciclo da peça 2

Fonte: Elaborado pelo autor.

5.5.2.2 Execução com o Grafo de Conhecimento Modificado

Na segunda etapa, o Grafo de Conhecimento foi alterado para simular a substituição do atuador de descarte, e a URI de seu *endpoint* foi modificada. Ao executar novamente o mesmo cenário de teste com o agente inalterado, a sequência de eventos observada no simulador foi funcionalmente idêntica à apresentada na Tabela 3.

Esta validação qualitativa demonstra de forma inequívoca o desacoplamento proporcionado pela arquitetura semântica. O agente foi capaz de descobrir a nova URI ao consultar o grafo pela capacidade funcional do atuador, extrair a nova informação de interação de sua *affordance* e realizar a chamada HTTP com sucesso. A capacidade de se adaptar a uma mudança na infraestrutura física atende diretamente ao objetivo geral do projeto de reduzir o acoplamento entre o agente e o hardware.

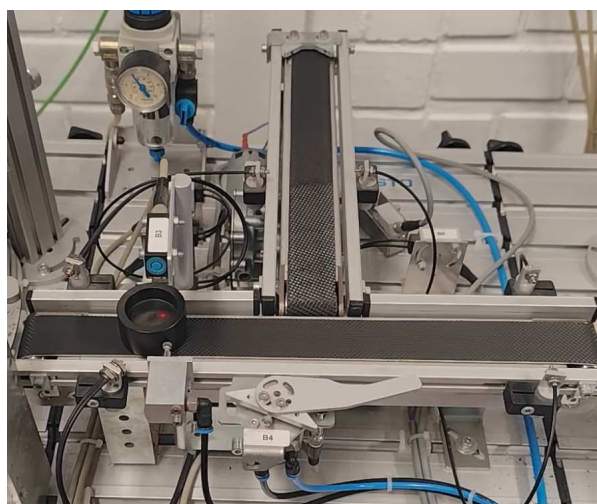
5.5.3 Validação em Ambiente Físico

Para complementar a validação em ambiente simulado e demonstrar a aplicabilidade da solução no mundo real, o sistema foi executado na planta física do LAI.

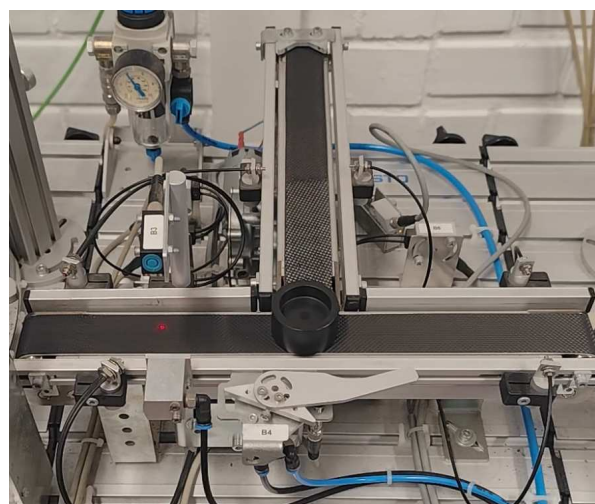
A transição do ambiente simulado para o físico exigiu apenas uma alteração de configuração: a atualização do endereço do servidor OPC UA na variável de ambiente `OPCUA_ENDPOINT_SEPARATING` para apontar para o CLP da bancada *Separating*. É fundamental ressaltar que nenhuma alteração foi realizada no Grafo de Conhecimento ou no código-fonte do agente, comprovando a portabilidade da solução.

O agente foi instruído a executar a mesma sequência de tarefas: processar uma peça com orientação correta e, em seguida, uma peça invertida. O comportamento da planta foi registrado em vídeo e os momentos chave de cada ciclo são apresentados nas figuras a seguir.

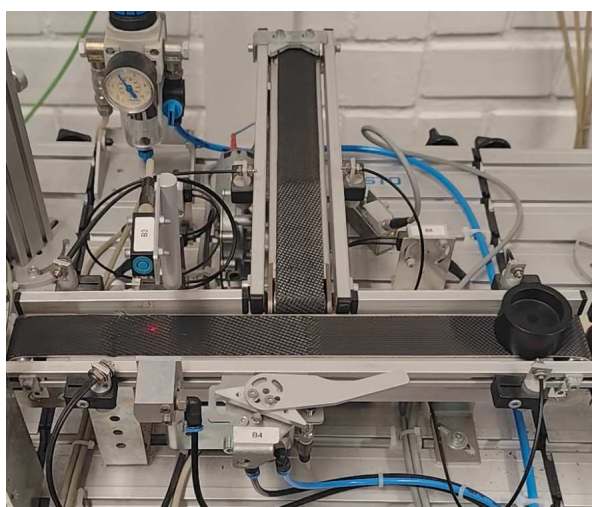
A Figura 12 demonstra o ciclo de passagem da peça correta. Após a inspeção (a), o agente identifica que a peça não deve ser descartada, libera a trava (b) e permite que a peça continue seu fluxo na esteira principal (c).



(a) Peça posicionada para inspeção.



(b) Trava liberada.

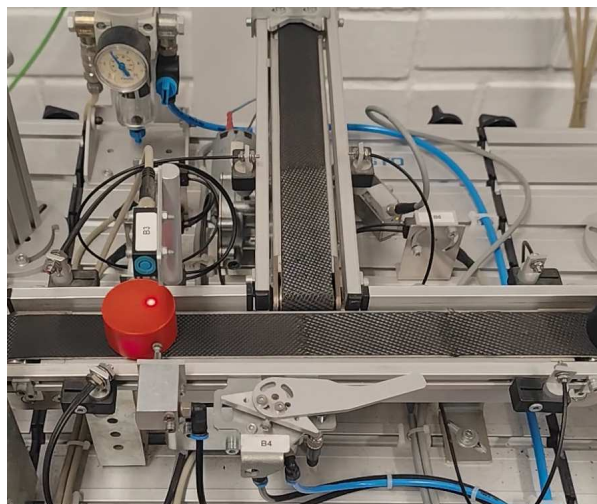


(c) Peça prosseguindo no fluxo.

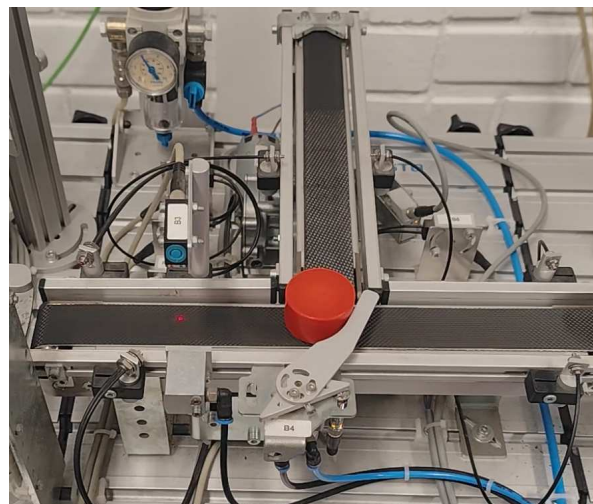
Figura 12 – Sequência de eventos do ciclo de passagem na planta física.

De forma análoga, a Figura 13 ilustra o ciclo de descarte. Após a peça ser

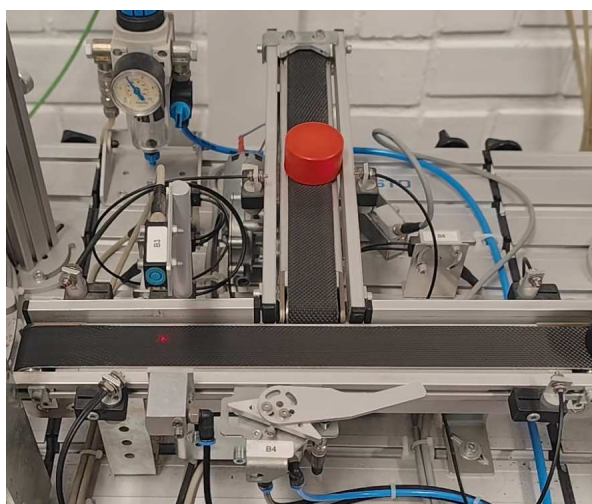
posicionada e travada para inspeção (a), o agente, ao receber o valor do sensor de orientação, corretamente aciona o atuador de descarte (b), que transfere a peça para a esteira secundária (c).



(a) Peça posicionada para inspeção.



(b) Atuador de descarte acionado.



(c) Peça na esteira de descarte.

Figura 13 – Sequência de eventos do ciclo de descarte na planta física.

O sucesso da execução na planta física, utilizando o mesmo software validado no ambiente simulado, comprova de a robustez da arquitetura proposta. A solução demonstrou ser capaz de controlar o hardware real, validando a eficácia de toda a cadeia de comunicação, desde a camada de inteligência até a camada física.

5.5.4 Análise Crítica e Limitações da Solução

A análise comparativa evidencia que a arquitetura proposta soluciona o problema do acoplamento nominal, identificado na abordagem anterior, ao substituir a dependência de identificadores estáticos pela descoberta de recursos baseada em capacidades funcionais. A principal vantagem desta abordagem é o ganho direto em

flexibilidade e robustez a mudanças, uma capacidade crucial em cenários de Indústria 4.0, onde as linhas de produção são dinâmicas e a reconfiguração de hardware é uma necessidade operacional constante.

No entanto, é fundamental reconhecer as limitações da solução. A principal desvantagem é a maior complexidade inicial na modelagem. A criação de um Grafo de Conhecimento semanticamente correto, que integra múltiplas ontologias, exige um esforço de design e um conhecimento técnico significativamente maiores do que a abordagem anterior, que representava a planta através de um único documento TD. Essencialmente, esta abordagem transfere a complexidade da lógica de programação do agente para a representação do conhecimento, demandando um maior investimento inicial na fase de projeto.

A limitação de escopo mais importante da solução é o acoplamento de processo. A arquitetura proposta resolve com sucesso o acoplamento nominal, pois o agente descobre dinamicamente a identidade dos dispositivos e os detalhes de sua interface de comunicação. Contudo, a lógica que governa o fluxo de trabalho — a sequência de funções a serem executadas e a ordem entre elas — permanece explicitamente codificada nos planos do agente. Consequentemente, mudanças radicais no processo industrial, que alterem a ordenação das tarefas, ainda exigiriam uma reprogramação da lógica do agente.

A verificação dos requisitos e a validação empírica do desacoplamento semântico, detalhadas nesta seção, comprovam que a solução implementada atende aos objetivos propostos e resolve o problema central que motivou este trabalho. A análise crítica demonstrou as vantagens da abordagem em termos de flexibilidade e robustez, ao mesmo tempo em que reconheceu suas limitações, como a complexidade inicial da modelagem e o acoplamento de processo remanescente. Com a implementação detalhada e os resultados analisados, o capítulo seguinte apresentará a conclusão geral do trabalho, onde serão discutidos os impactos e as direções para futuras pesquisas.

6 CONCLUSÃO

O presente trabalho partiu do problema do acoplamento nominal em sistemas de automação, uma limitação identificada em uma arquitetura anterior baseada em WoT, onde o agente de controle dependia de identificadores estáticos para interagir com os dispositivos da planta. A solução proposta foi uma arquitetura que introduz uma camada de conhecimento semântico, materializada como um Grafo de Conhecimento em RDF. A metodologia baseou-se na composição de ontologias padrão (SOSA, SAREF, AI4Industry, TD) para criar uma descrição rica e funcional da planta. A implementação desta arquitetura foi realizada através de um construtor programático em Java com Apache Jena, utilizando como cliente um agente de software desenvolvido com a plataforma JaCaMo. O principal resultado, validado empiricamente, foi a demonstração de que o agente de software é capaz de se adaptar a mudanças na infraestrutura de hardware, como a substituição de um atuador, sem qualquer modificação em seu código-fonte.

Com isso, os objetivos inicialmente traçados foram atingidos: foi desenvolvida e validada uma arquitetura semântica completa, desde a modelagem do Grafo de Conhecimento até o seu consumo por um agente de software. A principal contribuição técnica deste trabalho é a definição e validação de um padrão de modelagem que integra as ontologias SOSA, SAREF e TD para habilitar a descoberta de recursos por suas capacidades funcionais, e não por seus identificadores.

O impacto prático desta contribuição se manifesta em duas frentes. Para o LAI, o projeto entrega uma plataforma de pesquisa replicável, composta pelo modelo semântico, pelo simulador da planta e pelo ambiente containerizado, que habilita o desenvolvimento e a validação de agentes autônomos focados em metas de alto nível. Para o contexto mais amplo da Indústria 4.0, o trabalho reforça a viabilidade de sistemas mais resilientes e de menor custo de manutenção, onde a reconfiguração da planta, uma necessidade operacional constante, pode ser realizada com impacto mínimo no software de controle.

6.1 TRABALHOS FUTUROS

As limitações e a análise de resultados, discutidas no capítulo anterior, abrem diversas e promissoras direções para trabalhos futuros. A sugestão mais direta é a aplicação da metodologia de modelagem nas outras cinco bancadas da linha de produção FESTO, o que permitiria validar a escalabilidade da arquitetura e explorar o raciocínio do agente sobre um fluxo de produção completo e mais complexo.

Para endereçar a limitação do acoplamento de processo, uma linha de pesquisa promissora seria a modelagem da própria lógica sequencial no Grafo de Conhecimento. Utilizando padrões como BPMN-RDF ou explorando a ontologia SAREF em

maior profundidade, seria possível descrever o fluxo de trabalho semanticamente. Tal representação permitiria que a sequência de operações se tornasse uma fonte de conhecimento consultável, desacoplando-a da lógica interna dos planos do agente e abrindo caminho para sistemas de controle ainda mais flexíveis e reconfiguráveis.

Finalmente, a utilidade da arquitetura pode ser explorada em cenários que envolvam maior heterogeneidade tecnológica, um desafio comum em ambientes industriais dinâmicos. Poderia ser investigada a integração transparente de dispositivos IoT nativos, que expõem suas próprias APIs. Ao descrevê-los semanticamente e adicioná-los ao grafo, seria possível validar a capacidade do agente de orquestrar um sistema híbrido, composto por componentes legados (controlados via OPC UA) e dispositivos modernos, utilizando a mesma lógica de descoberta funcional.

REFERÊNCIAS

BECKETT, David; BERNERS-LEE, Tim. **Turtle - Terse RDF Triple Language**. [S.l.], mar. 2011. Disponível em: <https://www.w3.org/TeamSubmission/turtle/>.

BERNERS-LEE, T.; FIELDING, R.; MASINTER, L. **Uniform Resource Identifier (URI): Generic Syntax**. [S.l.], jan. 2005. Disponível em: <https://datatracker.ietf.org/doc/html/rfc3986>. Acesso em: 23 maio 2024.

BERNERS-LEE, Tim; HENDLER, James; LASSILA, Ora. The Semantic Web. **Scientific American**, v. 284, n. 5, p. 34–43, maio 2001.

BORDINI, Rafael H.; HÜBNER, Jomi Fred; WOOLDRIDGE, Michael. Programming Multi-Agent Systems in AgentSpeak using Jason. **John Wiley & Sons**, 2007. DOI: 10.1002/9780470061848.

BUCHGEHER, Georg; GABAUER, David; MARTINEZ-GIL, Jorge; EHRLINGER, Lisa. Knowledge Graphs in Manufacturing and Production: A Systematic Literature Review. **IEEE Access**, v. 9, p. 55537–55554, 2019. DOI: 10.1109/ACCESS.2021.3070395.

CHAMPIN, Pierre-Antoine. **JSON-LD and RDF**. [S.l.], jun. 2014. Disponível em: <https://www.w3.org/TR/json-ld-rdf/>.

CHARPENAY, Victor; KÄBISCH, Sebastian. On Modeling the Physical World as a Collection of Things: The W3C Thing Description Ontology. In: THE Semantic Web: 17th International Conference, ESWC 2020. [S.l.]: Springer, 2020. p. 599–615. DOI: 10.1007/978-3-030-49461-2_35.

CHARPENAY, Victor; ZIMMERMANN, Antoine; LEFRANÇOIS, Maxime; BOISSIER, Olivier. Hypermedea: A Framework for Web (of Things) Agents. In: COMPANION Proceedings of the Web Conference 2022. [S.l.: s.n.], 2022. p. 176–179. DOI: 10.1145/3487553.3524243.

COMPTON, Michael et al. The SSN ontology of the W3C semantic sensor network incubator group. **Journal of Web Semantics**, v. 17, p. 25–32, 2012. DOI: 10.1016/j.websem.2012.05.003.

DOCKER, INC. **Docker: Accelerated, Containerized Application Development**. [S.l.: s.n.], 2025. Website. Disponível em: <https://www.docker.com/>. Acesso em: 8 jul. 2025.

ETSI. **SmartM2M; Smart Applications; Reference Ontology (SAREF)**. [S.l.], 2022. Disponível em: https://www.etsi.org/deliver/etsi_ts/103200_103299/103264/04_01_01_60/ts_103264v040101p.pdf. Acesso em: 22 maio 2024.

HALLER, Armin et al. **Semantic Sensor Network Ontology**. [S.l.], jan. 2017. Disponível em: <https://www.w3.org/TR/vocab-ssn/>.

HERMANN, Mario; PENTEK, Tobias; OTTO, Boris. Design Principles for Industrie 4.0 Scenarios. In: 2016 49th Hawaii International Conference on System Sciences (HICSS). [S.l.]: IEEE, 2016. p. 3928–3937. DOI: 10.1109/HICSS.2016.488.

Ji, Shaoxiong; PAN, Shirui; CAMBRIA, Erik; MARTTINEN, Pekka; YU, Philip S. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. **IEEE Transactions on Neural Networks and Learning Systems**, v. 33, n. 2, p. 494–514, 2021. DOI: 10.1109/TNNLS.2021.3070843.

JIRKOVSKY, Vaclav; OBITKO, Marek; MARIK, Vladimir. Understanding Data Heterogeneity in the Context of Cyber-Physical Systems Integration. **IEEE Transactions on Industrial Informatics**, v. 13, n. 2, p. 660–667, 2017. DOI: 10.1109/TII.2016.2596101.

KÄBISCH, Sebastian; KOSTER, Michael; KORKAN, Ege; LAGALLY, Michael; MATSUKURA, Ryuichi; MCCOOL, Michael; TOUMURA, Kunihiro. **Web of Things (WoT) Thing Description**. [S.l.], jun. 2023. Disponível em: <https://www.w3.org/TR/wot-thing-description11/>.

KAGERMANN, Henning; WAHLSTER, Wolfgang; HELBIG, Johannes. **Recommendations for implementing the strategic initiative INDUSTRIE 4.0**. Frankfurt/Main, Germany, 2013.

KLYNE, Graham; CARROLL, Jeremy J. **Resource Description Framework (RDF): Concepts and Abstract Syntax**. [S.l.], 2004. Disponível em: <https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.

KORKAN, Ege; KAEBISCH, Sebastian; STEINHORST, Sebastian. Streamlining IoT system development with open standards. **it - Information Technology**, v. 62, n. 5-6, p. 215–226, 2020. DOI: 10.1515/itit-2020-0016.

LEITÃO, Paulo. Agent-based distributed manufacturing control: A state-of-the-art survey. **Engineering Applications of Artificial Intelligence**, v. 22, n. 7, p. 979–991, 2009. DOI: 10.1016/j.engappai.2008.09.005.

LEITÃO, Paulo; KARNOUSKOS, Stamatis; RIBEIRO, Luis; LEE, Jay; STRASSER, Thomas; COLOMBO, Armando W. Smart agents in industrial cyber-physical systems. **Proceedings of the IEEE**, v. 104, n. 5, p. 1086–1101, 2016. DOI: 10.1109/JPROC.2016.2521931.

MONOSTORI, László; KÁDÁR, Botond; BAUERNHANSL, Thomas et al. Cyber-physical systems in manufacturing. **CIRP Annals**, v. 65, n. 2, p. 621–641, 2016. DOI: 10.1016/j.cirp.2016.06.005.

NARDIN, L. G. **2024 summer school on AI for industry 4.0**. [S.l.: s.n.], 2024. Summer school on AI technologies for trust, interoperability, autonomy and resilience in industry. Disponível em: <https://ci.mines-stetienne.fr/ai4industry/2024/>.

RAO, Anand S.; GEORGEFF, Michael P. BDI Agents: From Theory to Practice. In: PROCEEDINGS of the First International Conference on Multi-Agent Systems. [S.l.: s.n.], 1995. p. 312–319.

SPORNY, Manu; LONGLEY, Dave; KELLOGG, Gregg; LANTHALER, Markus; LINDSTRÖM, Niklas. **JSON-LD 1.1: A JSON-based Serialization for Linked Data**. [S.l.], jul. 2020. Disponível em: <https://www.w3.org/TR/json-ld11/>.

THE APACHE SOFTWARE FOUNDATION. **Apache Jena - Documentation**. [S.l.: s.n.], 2025. Website. Disponível em: https://jena.apache.org/getting_started/index.html. Acesso em: 6 jul. 2025.

W3C. **JSON-LD 1.1: A JSON-based Serialization for Linked Data**. [S.l.], jul. 2020a. Disponível em: <https://www.w3.org/TR/json-ld11/>.

W3C. **RDF 1.1 Concepts and Abstract Syntax**. [S.l.], fev. 2014a. Acessado em: 2025. Disponível em: <https://www.w3.org/TR/rdf11-concepts/>.

W3C. **RDF 1.1 Primer**. [S.l.], jun. 2014b. Acessado em: 2025. Disponível em: <https://www.w3.org/TR/rdf11-primer/>.

W3C. **Web of Things (WoT) Architecture**. [S.l.], abr. 2020b. Disponível em: <https://www.w3.org/TR/wot-architecture/>.

W3C. **Web of Things (WoT) Thing Description**. [S.l.], abr. 2020c. Disponível em: <https://www.w3.org/TR/wot-thing-description/>.

ZENG, Deze; GUO, Song; CHENG, Zixue. The Web of Things: A Survey. **Journal of Communications**, v. 6, n. 6, p. 424–438, 2011. DOI: 10.4304/jcm.6.6.424-438.

Apêndices

APÊNDICE A – CÓDIGO FONTE DO SERVIDOR *LINKED DATA*

```

1 package br.ufsc.agents_in_charge;
2
3 import java.io.IOException;
4 import java.io.OutputStream;
5 import java.util.HashSet;
6 import java.util.Set;
7 import java.util.regex.Matcher;
8 import java.util.regex.Pattern;
9
10 import org.apache.jena.query.Dataset;
11 import org.apache.jena.query.ReadWrite;
12 import org.apache.jena.rdf.model.Model;
13 import org.apache.jena.rdf.model.ModelFactory;
14 import org.apache.jena.rdf.model.RDFNode;
15 import org.apache.jena.rdf.model.Resource;
16 import org.apache.jena.rdf.model.Statement;
17 import org.apache.jena.rdf.model.StmtIterator;
18 import org.apache.jena.riot.Lang;
19 import org.apache.jena.riot.RDFDataMgr;
20 import org.apache.jena.tdb2.TDB2Factory;
21 import org.eclipse.jetty.server.Server;
22 import org.eclipse.jetty.servlet.ServletContextHandler;
23 import org.eclipse.jetty.servlet.ServletHolder;
24
25 import jakarta.servlet.ServletException;
26 import jakarta.servlet.http.HttpServlet;
27 import jakarta.servlet.http.HttpServletRequest;
28 import jakarta.servlet.http.HttpServletResponse;
29
30 /**
31  * Servidor HTTP que expoe os dados RDF com content negotiation
32  */
33 public class LinkedDataServer {
34     private final String tdbDirectory;
35     private final int port;
36     private final Pattern resourcePattern = Pattern.compile("/kg/(.+)");
37
38     public LinkedDataServer(String tdbDirectory, int port) {
39         this.tdbDirectory = tdbDirectory;
40         this.port = port;
41     }
42
43     /**
44      * Inicia o servidor HTTP
45      */

```

```
46 public void start() throws Exception {
47     System.out.println("Iniciando servidor na porta " + port);
48
49     // Criar servidor Jetty
50     Server server = new Server(port);
51
52     ServletContextHandler context = new
53     ServletContextHandler(ServletContextHandler.SESSIONS);
54     context.setContextPath("/");
55     server.setHandler(context);
56
57     // Adicionar servlet para atender as requisicoes
58     context.addServlet(new ServletHolder(new LinkedDataServlet()),
59     "/*");
60
61     // Iniciar o servidor
62     server.start();
63     System.out.println("Servidor iniciado com sucesso em
64     http://localhost:" + port);
65     System.out.println("Exemplo de recurso: http://localhost:" + port +
66     "/kg/HeightSensor");
67     System.out.println("Pressione Ctrl+C para encerrar o servidor");
68     server.join();
69 }
70
71 /**
72  * Servlet para lidar com as requisicoes HTTP
73  */
74 private class LinkedDataServlet extends HttpServlet {
75     @Override
76     protected void doGet(HttpServletRequest req, HttpServletResponse
77     resp)
78         throws ServletException, IOException {
79
80         String path = req.getRequestURI();
81
82         if (path.startsWith("/kg/vocabulary")) {
83             serveVocabularyGraph(req, resp);
84             return;
85         }
86
87         if (path.equals("/kg") || path.equals("/kg/")) {
88             serveFullGraph(req, resp);
89             return;
90         }
91
92         Matcher matcher = resourcePattern.matcher(path);
```

```
88     if (!matcher.matches()) {
89         resp.setStatus(HttpServletResponse.SC_NOT_FOUND);
90         resp.getWriter().println("Invalid resource path. Path must
91             start with /kg/");
92         return;
93     }
94
95     String resourceId = matcher.group(1);
96     String resourceUri = Vocabulary.BASE_URI + resourceId;
97
98     Dataset dataset = TDB2Factory.connectDataset(tdbDirectory);
99     try {
100         dataset.begin(ReadWrite.READ);
101         Model model = dataset.getDefaultModel();
102         Resource resource = model.getResource(resourceUri);
103
104         if (!model.containsResource(resource)) {
105             resp.setStatus(HttpServletResponse.SC_NOT_FOUND);
106             resp.getWriter().println("Resource not found: " +
107                 resourceUri);
108             return;
109         }
110
111         Model resourceModel = extractMeaningfulSubgraph(model,
112             resource);
113
114         resourceModel.setNsPrefixes(model.getNsPrefixMap());
115         serverRdfResponse(req, resp, resourceModel);
116
117     } finally {
118         if (dataset != null) {
119             dataset.end();
120             dataset.close();
121         }
122     }
123
124     /**
125     * Serves the vocabulary definitions (TBox) from the TDB dataset.
126     */
127     private void serveVocabularyGraph(HttpServletRequest req,
128         HttpServletResponse resp) throws IOException {
129         Dataset dataset = TDB2Factory.connectDataset(tdbDirectory);
130         try {
131             dataset.begin(ReadWrite.READ);
132             Model fullModel = dataset.getDefaultModel();
```

```

131      // --- Filtering Logic for Vocabulary ---
132      Model vocabModel = ModelFactory.createDefaultModel();
133      vocabModel.setNsPrefixes(fullModel.getNsPrefixMap());
134
135      StmtIterator iter = fullModel.listStatements();
136      while (iter.hasNext()) {
137          Statement stmt = iter.next();
138          Resource subject = stmt.getSubject();
139
140          // Check if the subject's namespace IS the vocabulary
141          // namespace
142          if (subject.getNamespace() != null &&
143              subject.getNamespace().equals(Vocabulary.VOCAB_URI)) {
144              vocabModel.add(stmt);
145          }
146      }
147      // --- End of Filtering Logic ---
148
149      serverRdfResponse(req, resp, vocabModel);
150
151      } finally {
152          if (dataset != null) {
153              dataset.end();
154              dataset.close();
155          }
156      }
157
158      /**
159       * Serves the entire default graph from the TDB dataset.
160       */
161      private void serveFullGraph(HttpServletRequest req,
162      HttpServletResponse resp) throws IOException {
163          Dataset dataset = TDB2Factory.connectDataset(tdbDirectory);
164          try {
165              dataset.begin(ReadWrite.READ);
166              Model model = dataset.getDefaultModel();
167              serverRdfResponse(req, resp, model);
168          } finally {
169              if (dataset != null) {
170                  dataset.end();
171                  dataset.close();
172              }
173          }
174      }
175
176      /**

```

```

175     * Handles content negotiation and writes the RDF model to the
176     response.
177     */
178     private void serveRdfResponse(HttpServletRequest req,
179     HttpServletResponse resp, Model model) throws IOException {
180         String acceptHeader = req.getHeader("Accept");
181         Lang outputLang = Lang.TURTLE; // Default to Turtle
182         String contentType = "text/turtle;charset=UTF-8";
183
184         if (acceptHeader != null) {
185             if (acceptHeader.contains("application/ld+json")) {
186                 outputLang = Lang.JSONLD;
187                 contentType = "application/ld+json;charset=UTF-8";
188             } else if (acceptHeader.contains("application/rdf+xml")) {
189                 outputLang = Lang.RDFXML;
190                 contentType = "application/rdf+xml;charset=UTF-8";
191             }
192             // Add more formats if needed (e.g., N-TRIPLES)
193         }
194
195         resp.setContentType(contentType);
196         resp.setCharacterEncoding("UTF-8"); // Important for special
197         characters
198         OutputStream out = resp.getOutputStream();
199         RDFDataMgr.write(out, model, outputLang);
200     }
201 }
202
203 /**
204  * Extracts properties of the central resource and full descriptions
205  * of any
206  * blank nodes connected to it.
207  * This includes:
208  * 1. All properties of the central resource.
209  * 2. The full description of any blank nodes connected to the
210  * central resource.
211  *
212  * @param sourceModel The full knowledge graph.
213  * @param centerNode The resource to describe.
214  * @return A new Model containing the subgraph.
215  */
216 private Model extractMeaningfulSubgraph(Model sourceModel, Resource
217 centerNode) {
218     Model targetModel = ModelFactory.createDefaultModel();
219
220     Set<Resource> visited = new HashSet<>();

```

```
216 StmtIterator centerIter = sourceModel.listStatements(centerNode,
217 null, (RDFNode) null);
218 while (centerIter.hasNext()) {
219     Statement stmt = centerIter.next();
220     targetModel.add(stmt);
221     RDFNode object = stmt.getObject();
222     if (object.isAnon()) {
223         extractRecursive(sourceModel, object, targetModel, visited);
224     }
225 }
226 return targetModel;
227 }
228
229 private void extractRecursive(Model source, RDFNode node, Model
target, Set<Resource> visited) {
230     if (!node.isResource()) {
231         return;
232     }
233
234     Resource resource = node.asResource();
235
236     if (!resource.isAnon()) {
237         return;
238     }
239
240     if (visited.contains(resource)) {
241         return;
242     }
243
244     visited.add(resource);
245
246     StmtIterator subjIter = source.listStatements(resource, null,
(RDFNode) null);
247     while (subjIter.hasNext()) {
248         Statement stmt = subjIter.next();
249         target.add(stmt);
250         RDFNode object = stmt.getObject();
251         if (object.isAnon()) {
252             extractRecursive(source, object, target, visited);
253         }
254     }
255 }
256 }
```

Código A.1 – Implementacao da classe LinkedDataServer