UNIVERSIDADE FEDERAL DO AMAZONAS

FACULDADE DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Rafael da Silva Mendonça

PLATAFORMA DIDÁTICA PARA DESENVOLVIMENTO DE SISTEMAS MULTIAGENTE

Manaus

2016

Rafael da Silva Mendonça

PLATAFORMA DIDÁTICA PARA DESENVOLVIMENTO DE SISTEMAS MULTIAGENTE

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Orientador: Prof. Dr. André Luiz Duarte Cavalcante

Manaus

2016

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

M539p

Mendonça, Rafael da Silva
Plataforma Didática para Desenvolvimento de Sistemas Multiagente / Rafael da Silva Mendonça. 2016
80 f.: il. color; 31 cm.

Orientador: Prof. Dr. André Luiz Duarte Cavalcante
Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Amazonas.

1. agentes mecatrônicos. 2. Plataforma de Desenvolvimento. 3. Simulador. 4. Multiagente. 5. Máquina de Estados Finita. I. Cavalcante, Prof. Dr. André Luiz Duarte II. Universidade Federal do Amazonas III. Título

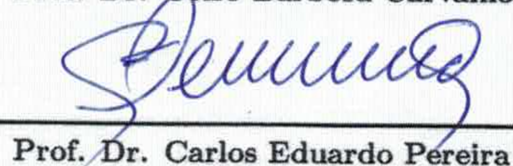
Rafael da Silva Mendonça

PLATAFORMA DIDÁTICA PARA DESENVOLVIMENTO DE SISTEMAS MULTIAGENTE

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Prof. Dr. André Luiz Duarte Cavalcante

Prof. Dr. Celso Barbosa Carvalho



Prof. Dr. Carlos Eduardo Pereira

Manaus
2016

"A sabedoria é resplandecente, não murcha,
mostra-se facilmente para aqueles que a amam.
Ela se deixa encontrar por aqueles que a buscam.
(Bíblia Sagrada, Sabedoria 6, 12)

Resumo

Este trabalho é a proposta de uma plataforma didática para o desenvolvimento e o aprendizado de agentes inteligentes aplicados à manufatura. Além disso, é proposto um Simulador para agentes mecatrônicos que visa a reprodução do comportamento temporal de um *hardware* mecatrônico através da definição de sua máquina de estados finita. Essa plataforma foi validada por uma aplicação real e o Simulador pela comparação qualitativa com essa aplicação. O comportamento do sistema real e do sistema simulado são próximos, de forma que a substituição da máquina simulada pela real é transparente para o restante do sistema.

Palavras-chave: agentes mecatrônicos. Plataforma de Desenvolvimento. Multiagente. Simulador. Máquina de Estados Finita.

Abstract

This work is the proposal of a didactic platform for the development and learning of intelligent agents applied to manufacturing. Furthermore, it is proposed a simulator for mechatronic agents aimed at reproducing the temporal behavior of a mechatronic hardware by defining a finite state machine. This platform has been validated by a real application and the Simulator by qualitative comparison with that application. The real system behavior and the simulated system behavior are close, so that the replacement of the simulated real machine is transparent to the rest of the system.

Keywords: Mechatronic Agents. Simulator. Development Platform. Multiagent, Finite State Machine.

Lista de ilustrações

Figura 1 – FIPA *Request Interaction Protocol* 7
Figura 2 – Interface Gráfica do JADE 9
Figura 3 – Plataforma e Contêineres 10
Figura 4 – Ciclo de Vida do Agente 11
Figura 5 – Ciclo de Execução de um Agente 12
Figura 6 – Sistema RMS 16
Figura 7 – Relação entre *framework*, Aplicação e Bibliotecas. 21
Figura 8 – EMAS - Sistema Multiagentes Evolutivo. 25
Figura 9 – Comunicação entre agentes usando ontologia. 27
Figura 10 – IADE (IDEAS Agent Development Environment) 29
Figura 11 – Camadas da Arquitetura EPSCore 37
Figura 12 – Proposta do Simulador EPS. 39
Figura 13 – Máquina de Estados Genérica 41
Figura 14 – Funcionamento do Simulador EPS. 42
Figura 15 – *Skills* definidos na arquitetura EPSCore. 44
Figura 16 – Ontologia para EPS. 45
Figura 17 – Classe UML - Ontologia para EPS - *BeanOntology* e *AgentActions*. . . . 47
Figura 18 – Classes da Arquitetura EPSCore 50
Figura 19 – Diagrama de Classes em UML do YPA. 51
Figura 20 – Diagrama de classes em UML do MRA. 52
Figura 21 – Máquina de estados do agente produto. 53
Figura 22 – Interface do OrderAgent. 54
Figura 23 – Aplicação SIAPE 55
Figura 24 – Diagrama de Classe parcial da aplicação SIAPE 56
Figura 25 – Decodificação do XML 60
Figura 26 – Classe para Instanciação: InstanciaMA 62
Figura 27 – Classe para Instanciação: SkillCall 63
Figura 28 – Máquina de estados do módulo carimbador. 64
Figura 29 – Máquina de estados da Esteira. 64
Figura 30 – Trecho do XML dos Módulos Carimbadores (*Stamper*) 68
Figura 31 – Trecho do XML da Esteira (*Conveyor*) 68
Figura 32 – XML da chamada de Produto (*Skillcall*) 69
Figura 33 – Descrição das *skills* do Stamper 69
Figura 34 – Descrição das *skills* do Conveyor 70
Figura 35 – Ambiente para Simulação dos Agentes 70
Figura 36 – Proposta do Simulador EPS para Trabalhos Futuros. 76

Lista de tabelas

Tabela 1 – FIPA Parâmetros 8
Tabela 2 – Comparativo de MAS e SOA 25
Tabela 3 – Resumo dos Paradigmas 33
Tabela 4 – Comparação de Agentes das Arquiteturas 65
Tabela 5 – Comparativo do tempo de troca e acionamento dos módulos carimbadores 71
Tabela 6 – Comparativo do Tempo de produção dos anagramas no SIAPE e no Simulador EPS 73

Lista de abreviaturas e siglas

ACL *Agent Communication Language Specification*

AID *Agent IDentification*

AMI *Agent Machine Interface*

ADACOR *ADAptive holonic COntrol aRchitecture*

API *Application programming interface*

AMS *Agent Managment System*

BA *Broker Agent*

BMS *Bionic Manufacturing Systems*

CA *Coordination Agent*

CLA *Coalition Leader Agent*

CMgA *Cluster Manager Agent*

CoBASA *Coalition Based Approach for Shop Flor Agility*

DA *Deployment Agent*

DF *Directory Facilitator*

DNA *Deoxyribonucleic acid*

EAS *Evolvable Assembly Systems*

EMAS *Evolutionary Multi-Agent Systems*

EPS *Evolvable Production System*

EPSCore *Arquitetura EPS*

FIPA *Foundation for Inteligent Physical Agents*

FMS *Flexible Manufacturing System*

FSM *Finite State Machine*

GA *Gateway Agent*

HMS *Holonic Manufacturing Systems*

IAD *Inteligencia Artificial Distribuída*

IADE *IDEAS Agent Development Environment*

ID *Identification*

IDEAS *Instantly Deployable Evolvable Assembly Systems Project*

IHM Interface Homem-Máquina

JADE *JAVA Agent DEvelopment framwork*

MA *Motor Agent*

MAS *Multi-Agent Systems*

MRA *Mechatronic Resource Agent*

OWL *Web Ontology Language*

OWL-S *Semantic Markup for Web Services*

PA *Product Agent*

PROSA *Architecture for Holonic Manufacturing Systems*

RA *Resourse Agent*

RDF *Resource Description Framework*

RMS *Reconfigurable Manufacturing Systems*

RNA *Ribonucleic acid*

SAX *Simple API for XML*

ST *SkillTemplate*

SIAPE *Sistema Inteligente Ágil de Processo Evolutivo*

SOA *Service Oriented Architecture*

TA *Trasnport Agent*

TSA *Trasportation System Agents*

WSMO *Web Service Modeling Ontology*

XML *Extensible Markup Language*

YPA *Yellow Pages Agent*

Sumário

1 INTRODUÇÃO **1**
1.1 Definição do problema e hipótese de trabalho **2**
1.1.1 Definição do problema 2
1.1.2 Hipótese de trabalho 2
1.2 Objetivos Geral **2**
1.3 Objetivos Específicos **2**
1.4 Metodologia **3**
1.5 Contribuição **3**
1.6 Organização do Trabalho **4**

2 FUNDAMENTAÇÃO TEÓRICA **5**
2.1 Agentes **5**
2.1.1 *Foundation for Intelligent Physical Agents* (FIPA) 6
2.2 *Java Agent Development Framework* (JADE) **9**
2.3 Paradigmas **13**
2.3.1 *Bionic Manufacturing Systems* (BMS) 13
2.3.2 *Holonic Manufacturing Systems* (HMS) 14
2.3.3 *Reconfigurable Manufacturing Systems* (RMS) 15
2.3.4 *Evolvable Assembly System* (EAS) e *Evolvable Production Systems* (EPS) . 15
2.4 Emergência e Auto-organização **17**
2.5 Definições Básicas **18**
2.5.1 *Skill* 18
2.5.2 Máquina de Estados 19
2.5.3 XML 20
2.5.4 API, Biblioteca e *Framework* 21
2.6 Simulação de Sistemas **22**

3 ESTADO DA ARTE **24**
3.1 *Multi Agents System* (MAS) e *Service Oriented Architecture* (SOA) . **24**
3.1.1 *Evolutionary MAS* (EMAS) 24
3.2 Ontologia **26**
3.3 EPS **28**
3.4 Simulador **33**

4 PROPOSTA **35**
4.1 Proposta de Arquitetura **35**

4.2 **Proposta do Simulador** 39
4.3 **Etapas para o Desenvolvimento** 43

5 **IMPLEMENTAÇÃO** 44
5.1 ***Skill* na arquitetura EPSCore** 44
5.2 **Ontologia** 45
5.3 **Arquitetura EPSCore** 47
5.3.1 Definição dos Agentes 50
5.4 **Aplicação da Arquitetura EPSCore** 55
5.5 **Simulador EPS** 58
5.5.1 `Agent EPS Simulator` 59

6 **EXPERIMENTOS E RESULTADOS** 65
6.1 **Comparativo das arquiteturas** 65
6.2 **Experimentos no Simulador EPS** 66
6.2.1 Criando o XML da aplicação SIAPE 67
6.2.2 Criação da Máquina de Estados 68
6.3 **Comparação entre a Simulação e a Aplicação** 69

7 **ANÁLISE E CONCLUSÃO** 74

Referências 77

1 Introdução

A fabricação de produtos em média e grande escala sempre foi uma busca da manufatura. Entretanto, no decorrer dos anos, ela se depara com uma necessidade de mercado que exige produtos personalizados e altamente variados, o que significa menor escala. Tal acontecimento é chamado de *customização de massa*.

As linhas de montagem atendem com eficiência à produção em uma certa quantidade de um mesmo produto, mas com a crescente necessidade de customização, essa mesma linha não consegue responder dinamicamente às variação dos produtos. Assim, a customização representa um desafio de modernidade e dinamismo e exige que o sistema se torne flexível e tenha capacidade de se reconfigurar rapidamente (MEHRABI; ULSOY; KOREN, 1999).

Vários paradigmas surgiram em resposta a esse desafio e, dentre esses, destaca-se o *Evolvable Production System* (EPS), que traz o conceito de sistemas de produção evolutivos e o uso de agentes aplicados à manufatura. O paradigma EPS traz soluções para o desafio da customização por apresentar a proposta para sistemas auto-organizáveis e autoadaptáveis. Em um EPS, as ferramentas de manufatura são modulares e inteligentes.

Por exemplo, em um ambiente de montagem, onde o produto é altamente customizado, cada módulo é responsável por um tipo de serviço de montagem ou customização. O produto é formado através da interação desses módulos e em cada módulo existe um ou mais agentes responsáveis pelas tarefas. Assim, esse agrupamento de agentes é um sistema multiagente que soluciona problemas complexos através das técnicas de sistemas inteligentes distribuídos.

Em todo EPS, tais agentes são executados dentro de módulos de produção, que são equipamentos mecatrônicos, os quais são chamados de *agentes mecatrônicos*. Cada módulo mecatrônico integra mecânica, eletrônica, computação e comunicação em um único equipamento; assim, podem interagir com o mundo físico e ter a capacidade de reagir às mudanças que ocorrem no seu ambiente. Agentes mecatrônicos possuem características de cooperação, solicitação e execução de serviços (tarefas da produção), autonomia e interação com outros agentes. Essas características possibilitam ao sistema processos de auto-organização e de emergência.

Este trabalho é focado em um sistema multiagentes e propõe o desenvolvimento de uma arquitetura e um simulador para sistemas de manufatura baseados no paradigma EPS. Para isso, utiliza as implementações existentes como base; contudo, a proposta é didática, pois facilita o ensino da criação, instanciação e comunicação de agentes mecatrônicos e a execução de suas funcionalidades. Além disso, foi desenvolvida uma ontologia que auxilia na padronização da comunicação e um simulador. O simulador desenvolvido visa

a permitir ao desenvolvedor de EPS verificar problemas relacionados ao funcionamento temporal do sistema e à comunicação entre os agentes antes da construção físicas dos módulos mecatrônicos do EPS.

1.1 Definição do problema e hipótese de trabalho

1.1.1 Definição do problema

EPS tem alto custo financeiro e alta complexidade, o que dificulta sua aprendizagem e desenvolvimento de soluções. Há um lacuna na implementação das plataformas EPS existentes, que é a sua associação a um simulador temporal dos agentes mecatrônicos, de tal forma que o desenvolvedor possa ver as chamadas de *skills* sendo realizadas nos tempos adequados para o funcionamento do sistema real, mesmo não tendo ainda a planta física.

1.1.2 Hipótese de trabalho

Desenvolver uma arquitetura e um simulador EPS para sistemas mecatrônicos, em que seja possível criar os agentes e testá-los antes de implementar o sistema real, reduz os custos e a complexidade do sistema a ser desenvolvido, também por aproveitar os dados obtidos no simulador na implementação dos agentes no sistema real, bem como favorecer a visualização das interações entre os agentes e o aprendizado de EPS.

1.2 Objetivos Geral

Implementar uma arquitetura e um simulador para sistemas EPS, nos quais os agentes mecatrônicos possam ser criados e testados e, posteriormente, fazer a comparação dos resultados obtidos no simulador com o funcionamento do sistema mecatrônico real.

1.3 Objetivos Específicos

- Desenvolver uma implementação de uma arquitetura EPS simplificada;
- Comparar com as arquiteturas existentes;
- Aplicar a implementação a um sistema real;
- Criar um simulador para EPS;
- Criar uma aplicação no simulador, com base na aplicação real;
- Comparar os resultados temporais da aplicação simulada e da aplicação real.

1.4 Metodologia

-> Estudar as implementações EPS existentes;

-> Criar uma arquitetura e uma implementação baseados no paradigma EPS;

-> Elencar os requisitos de um sistema mecatrônico real;

-> Definir o escopo e as funcionalidades do Simulador EPS;

-> Desenvolver a arquitetura EPSCore e validá-la através de uma aplicação;

-> Desenvolver o simulador com base em um sistema mecatrônico real;

-> Comparar os dados do sistema real e do simulado;

-> Analisar os dados obtidos, apresentá-los e discuti-los.

1.5 Contribuição

As principais contribuições deste trabalho são o desenvolvimento da arquitetura EPSCore e do Simulador EPS. Na arquitetura, o principal legado é a definição simplificada de agentes mecatrônicos e não mecatrônicos e a divisão destes em camadas; e, na proposta do simulador, o legado é que o agente simulador é usado para simular o comportamento temporal dos agentes mecatrônicos de um sistema, sendo que a simulação não precisa explicitamente do hardware, ou seja, é possível sua simulação sem o sistema mecatrônico real.

A arquitetura EPSCore é uma proposta didática que visa ao ensino de agentes inteligentes aplicados à manufatura. Através dela é possível o desenvolvimento, a experimentação e o aperfeiçoamento das ferramentas e técnicas disponíveis para o desenvolvimento de agentes.

O Simulador EPS reproduz o comportamento dos agentes mecatrônicos do sistema real através dos agentes motores simulados que executam (faz chamadas de *skill*) em uma máquina de estados finita. O simulador visa principalmente à reprodução das características temporais dos agentes mecatrônicos, proporcionando, assim, o estudo do tempo de execução das ações ou funcionalidades dos agentes, a análise do funcionamento temporal do sistema como um todo, baseadas nas operações dos atuadores e sensores, as trocas de mensagens entre os agentes, e proporciona o estudo de possíveis melhorias no sistema real.

A fase inicial deste trabalho, no desenvolvimento da arquitetura EPSCore, gerou a seguinte publicação:

1. MENDONÇA, R. d. S.; CAVALCANTE, A. L. D.; JUNIOR, V. F. de L. EPSCore: A Didactic Open Architecture of an Evolvable Production System. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). [S.l.: s.n.], 2016. p. 1–4.

1.6 Organização do Trabalho

Este trabalho está organizado conforme a lista a seguir:

O capítulo 01 é a introdução que contém os objetivos e a contribuição;

No capítulo 02, são apresentadas as fundamentações teóricas, que contém os conceitos básicos de agentes, paradigmas da manufatura entre outros que serão usado neste trabalho, assim como definições de auto-organização e emergência, ontologia e sistemas mecatrônicos;

No capítulo 03, são apresentados o estado da arte, as arquiteturas atuais, as propostas de simulador existentes e de sistemas multiagentes;

No capítulo 04, é apresentada a proposta deste trabalho que é o desenvolvimento da arquitetura EPSCore e do Simulador EPS.

O capítulo 05 contém a implementação da proposta; nele serão apresentados todos os procedimentos necessários ao desenvolvimento da Arquitetura EPSCore, da aplicação desenvolvida para sua validação e o desenvolvimento do Simulador EPS.

O capítulo 06 apresentará os experimentos realizados e os resultados obtidos através do uso da plataforma EPSCore que compreende a arquitetura EPSCore e o Simulador EPS sendo usados para reproduzir o comportamento temporal de agentes mecatrônicos;

No capítulo 07, são feitas as análises e conclusões deste trabalho, discute e compara o resultado obtido e apresenta a proposta de trabalhos futuros.

2 Fundamentação Teórica

Este capítulo tem como objetivo introduzir os conceitos básicos referentes à teoria utilizada nesta dissertação. Serão apresentados conceitos dos paradigmas da manufatura, suas características principais, ontologia, emergência e auto-organização assim, como a descrição de agentes, de simulação e de sistemas multiagente.

2.1 Agentes

A união de conceitos de *Inteligência Artificial* e de *Processamento Distribuído* formam a *Inteligência Artificial Distribuída* (IAD), que tem como objetivo estudar as interações entre entidades inteligentes. Essas entidades são denominadas agentes e realizam atos de inteligência global por meio de processamento local e comunicação inter processos sendo que essas interações podem ser de cooperação, coexistência e competição.

Com o objetivo de definir o que são agentes, Wooldridge e Jennings (1995) apresentam uma definição geral para agentes, tendo apresentado uma *noção fraca* e outra *noção forte*.

A noção fraca de agentes é relacionada ao conceito de agentes computacionais com propriedades como autonomia, habilidade social e adaptabilidade:

- automonomia: relacionada aqui à autonomia de raciocínio, requer uma máquina de inferência e base de conhecimento sendo essencial em sistemas especialistas, de controle, robótica, jogos, agentes na internet entre outros;
- habilidade social: relacionada à comunicação e à cooperação entre agentes, é a junção de técnicas avançadas de sistemas distribuídos e inteligência artificial para formar protocolos, padrões de comunicação, raciocínio autônomo sobre crenças e confiabilidade do que está sendo comunicado, arquiteturas de interação social entre agentes e outros pontos;
- adaptabilidade: é a capacidade de adaptação a situações novas, às quais não foi fornecido todo o conhecimento necessário com antecedência o qual pode ser fornecido na forma de aprendizagem e/ou programação;

A noção forte de agentes é ligada à área de inteligência artificial e possui, além das propriedades citadas, noções relacionadas ao comportamento humano tais como conhecimento, crença, intenção e obrigação. Alguns atributos podem ser citados:

- mobilidade: é a habilidade de um agente para se transportar entre máquinas participantes de uma rede de computadores;
- veracidade: é a suposição que um agente não comunicará informações falsas de maneira intencional;
- racionalidade: é a suposição de que o agente sempre agirá para alcançar suas metas, e nunca contra seus objetivos, pelo menos na medida em que suas crenças o permitam.

Na arquitetura, os sistemas baseados em agentes podem apresentar duas abordagens: (WOOLDRIDGE, 2002)

- baseada em agentes reativos: o mecanismo de raciocínio está inscrito em seu código, isto é, é explícita, não necessitando de nenhuma inferência ou deliberação na escolha da ação a ser realizada;
- baseada em agentes deliberativos: são aqueles que possuem um mecanismo de raciocínio em que as ações a serem tomadas são escolhidas por um mecanismo de inferência e/ou deliberação.

Para o desenvolvimento de sistemas de manufatura, a principal vantagem do uso de agentes reativos é a sua resposta rápida, e a vantagem dos deliberativos é sua flexibilidade. Para agentes reativos, a principal desvantagem é a falta de flexibilidade no raciocínio, enquanto para os agentes deliberativos é o tempo de resposta ou reação às mudanças ambientais.

Além disso, ainda pode-se classificar os agentes em: *cognitivos* que podem raciocinar sobre as ações que executaram e planejar ações futuras; *reativos* que reagem a determinadas situações; *colaborativos* que têm finalidade de alcançar um objetivo global através de tarefas locais; e *competitivos* que competem para realizar tarefas com mais eficiência.

Quando um conjunto de agentes do mesmo tipo ou de tipos diferentes são implementados de forma que, conjuntamente, resolvam um problema, são chamados de sistemas Multiagentes (Multi-Agent Systems)(MAS).

2.1.1 *Foundation for Intelligent Physical Agents* (FIPA)

A *Foundation for Intelligent Physical Agents* (FIPA) define um conjunto de especificações que formam um modelo de referência para uma plataforma de agentes, isto é, protocolos, regras para comunicação, trasporte de mensagens e um conjunto de serviços que devem ser fornecidos ao se conceber sistemas multiagentes interoperáveis.

Segundo a FIPA um agente é "uma entidade que reside em um ambiente onde interpretam dados através de sensores, refletem eventos no ambiente e executam ações que

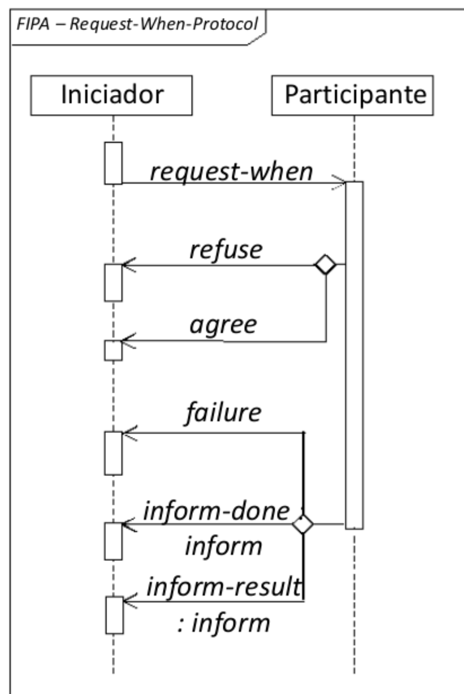


Figura 1 – FIPA *Request*. Fonte: (FIPA, 2016a)

produzem efeitos nesse mesmo ambiente. Um agente pode ser *software* puro ou *hardware* e *software* compostos." Assim, em linhas gerais, agentes são entidades de *software* autônomas que atuam em determinado ambiente de forma a interagir com este e com outros agentes, além de produzir ações e percepções sem requerer intervenções humanas constantes (SILVA, 2003).

Dentre o conjunto de especificações `FIPA`, utiliza-se,neste trabalho, as especificações de *Agent Communication Language specifications* (`ACL`) e do `FIPA` e o *FIPA Request Interaction protocol*.

Request Interaction protocol

O protocolo *FIPA Request Interaction Protocol* é usado na comunicação entre dois agentes em que um faz a solicitação do serviço e o outro responde ou não a essa solicitação. É uma relação caracterizada como cliente/servidor, contudo qualquer agente pode ser ambos cliente ou servidor. Quem inicia a comunicação é chamado de iniciador, e quem responde e executa o serviço é chamado de participante.

O *FIPA Request Interaction Protocol* permite que um agente, o Iniciador, solicite a outro, o Participante, para executar alguma ação, a mensagem REQUEST. O participante processa a solicitação e toma uma decisão de aceitar ou recusar o pedido. Se for tomada a decisão de recusar asolicitação, emite-se uma mensagem REFUSE. Se aceitar, emite-se uma mensagem AGREE. Se concorda em realizar o serviço, ato contínuo, realiza-o e, ao

Tabela 1 – FIPA Parâmetros *ACLMessage*. Fonte: (FIPA, 2016b)

Parâmetro	Categoria	Descrição
performative	Atos Comunicativos	Indica o tipo do ato comunicativo
sender	Participante na comunicação	Indica a identidade do remetente
receiver	Participante na comunicação	Indica a identidade dos destinatários
reply-to	Participante na comunicação	Indica que as mensagens serão direcionadas para o agente nomeado no reply-to.
content	Conteúdo da mensagem	Indica o conteúdo da mensagem
language	Descrição da conteúdo	Indica o idioma no qual é expresso.
encoding	Descrição da conteúdo	Denota a codificação específica
ontology	Descrição da conteúdo	Indica a ontologia que é utilizada
protocol	Controle de conversação	Protocolo de interação que o agente está empregando
conversation-id	Controle de conversação	Introduz um identificador de conversação
reply-with	Controle de conversação	Introduz uma expressão que será usada na resposta do agente
in-reply-to	Controle de conversação	Denota uma expressão que faz referência a uma ação
reply-by	Controle de conversação	Denota um tempo

final, emite um INFORM, em caso de o serviço ter sido realizado ou uma mensagem FAILURE, em caso de algum erro durante a realização do serviço. A Figura 1 apresenta o protocolo de interação FIPA Request:

Agent Communication Language (ACL)

As especificações FIPA ACL representam um conjunto de normas que se destinam a promover a interação dos agentes e os serviços disponibilizados. Uma mensagem FIPA ACL contém um conjunto de um ou mais parâmetros de mensagens.

O único parâmetro que é obrigatório em todas as mensagens ACL é o *performative*, embora espere-se que a maioria das mensagens também contenha *emissor*, *receptor* e *parâmetros de conteúdo* (FIPA, 2016b).

A tabela 1 mostra os parâmetros de mensagens FIPA ACL com uma pequena descrição para facilitar o entendimento e o uso destes. Cada especificação de ACL (mensagem de representação) contém descrições precisas da sintaxe de codificação de mensagens de ACL baseadas em XML, cadeias de textos e vários outros esquemas.

As mensagens FIPA são formadas preenchendo os campos de *destinatário*, *conteúdo* e *parâmetros da mensagem* e são associados às propriedades que auxiliam na comunicação. Se um agente não reconhece ou não é capaz de processar um ou mais dos valores do parâmetro ou parâmetros, a mensagem de retorno é uma "*NotUnderstood*".

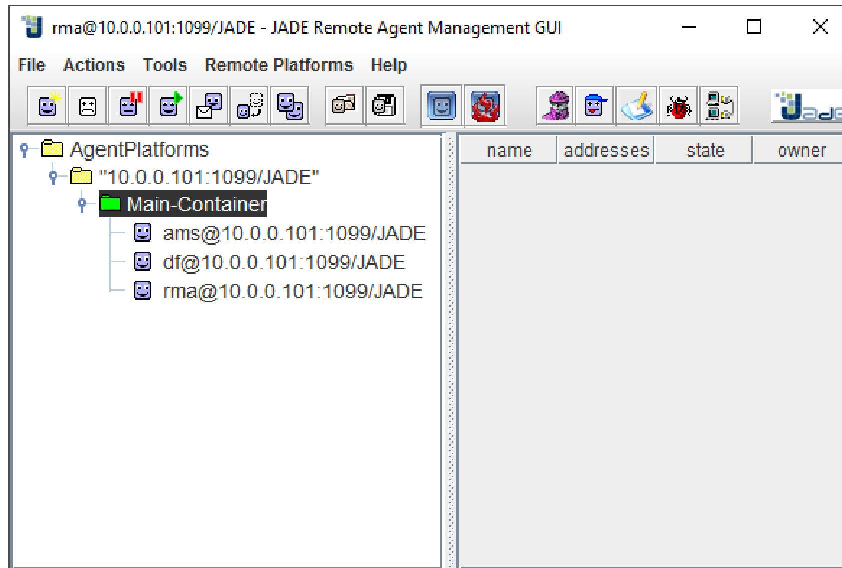


Figura 2 – Interface Gráfica da ferramenta Jade. Fonte: (FIPA, 2016a)

2.2 *Java Agent Development Framework* (JADE)

Java Agent Development Framework (JADE) é um *framework* para o desenvolvimento de aplicações multiagentes de acordo com as especificações da `FIPA`. Ou seja, é uma plataforma genérica para desenvolvimento de sistemas multiagente.

Um dos objetivos do `JADE` é o de simplificar o desenvolvimento de sistemas multiagentes. Assim, em conformidade com o padrão `FIPA`, é possível atender a serviços como o de páginas amarelas e transporte de mensagens.

Ao iniciar o ambiente `JADE` são criados os seguintes agentes especiais:

- *Agent Management System* (`AMS`) - é um agente de suporte que exerce o controle sobre o acesso e o uso da plataforma pelos agentes. O `AMS` é único na plataforma e todos os agentes devem ser nele registrados possibilitando, assim, a criação de uma lista de identificadores de agentes (`AID`), evitando que aconteça duplicação de nomes, situação esta, que causaria problemas na comunicação;

- *Directory Facilitator* (`DF`) - é responsável pelo serviço de páginas amarelas na plataforma, isto é, o registro, o encerrar registro, e a cancelamento e a busca de serviços e agentes. Este disponibiliza o `AID` dos agentes quando solicitado.

- *Remote Managment Agent* (`RMA`) [opcional] - é um agente responsável por um gerenciamento da plataforma, seus contêineres e dos demais agentes. A interface gráfica deste agente é apresentada na Figura 2:

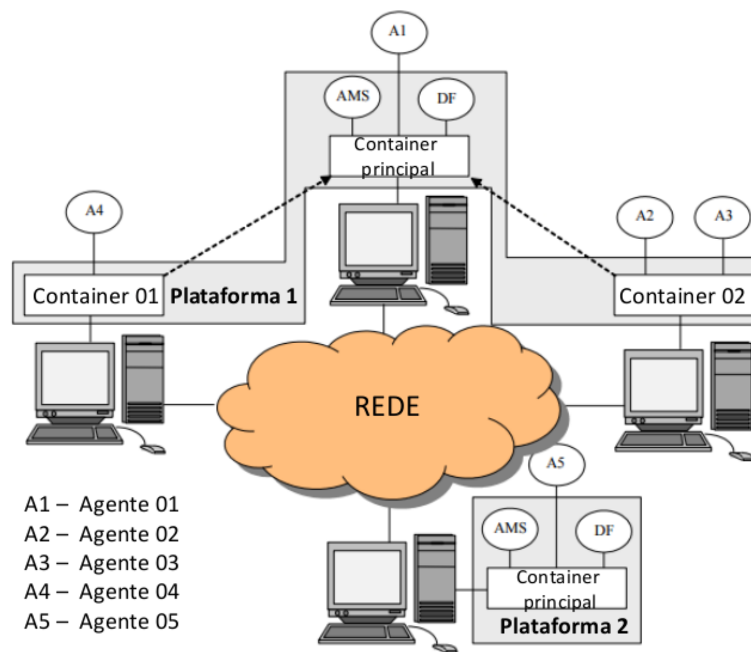


Figura 3 – Plataforma e Contêineres. Fonte: (CAIRE, 2016)

Uma plataforma `JADE` é um conjunto de contêineres (máquinas virtuais) que disponibilizam os serviços necessários para a execução e comunicação de agentes. A comunicação é otimizada dentro de cada *contêiner*. Para a comunicação entre *contêineres* e entre plataformas, é usado um protocolo próprio que otimiza a comunicação entre máquinas virtuais Java. Observe a Figura 3 para um esquema do ambiente JADE.

Neste trabalho, o framework `JADE` foi escolhido por fornecer uma plataforma adequada para o desenvolvimento das propostas, isto é, por fornecer diversos dos serviços necessários. Além disso, várias outras implementações de arquiteturas para EPS (BARATA; CAMARINHA-MATOS; ONORI, 2005; ONORI et al., 2012), foram feitas utilizando o mesmo ambiente o que permite uma melhor comparação qualitativa.

Na plataforma, dentro de uma máquina virtual `JADE`, um agente é criado e este é filho da classe `Agent`, a qual implementa uma *thread Java* que executa comportamentos continuamente. Essa execução segue a máquina de estados, apresentada na Figura 4, assim descrita:

-> estado ativo: após ser iniciado, o agente vai para o estado ativo e ali permanece até ser explicitamente colocado em espera ou suspenso (bloqueado);

-> estado em trânsito: o agente é colocado no estado em trânsito quando é solicitado fazer a migração entre as maquinas virtuais (contêineres Jade);

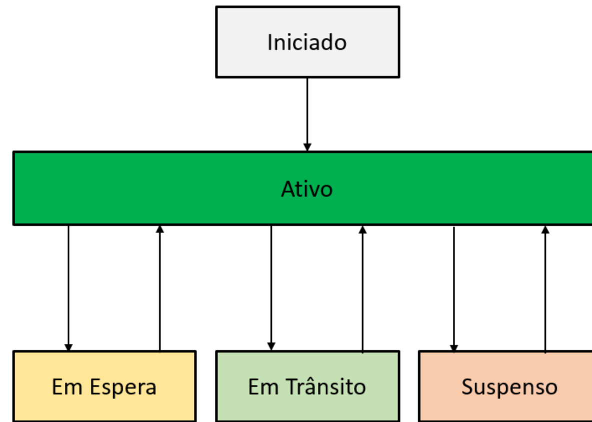


Figura 4 – Ciclo de Vida do Agente. Fonte: (CAIRE, 2016) com adaptações.

-> estado em espera: quando um agente está aguardando uma condição ou sinalização da JVM para continuar sua execução, então ele é colocado no estado de espera;

-> estado bloqueado (ou suspenso): há condições para que o agente seja explicitamente colocado em estado de bloqueio, entretanto, na maioria das vezes, o agente vai para o estado de bloqueio quando precisa aguardar um evento de comunicação.

O ciclo de execução de um agente é mostrado na Figura 5 onde é possível verificar que os comportamentos são executados continuamente, pela chamada do método *action()* do comportamento selecionado da fila de ativos. Um comportamento é uma entidade de execução de dentro do agente, modelada através da classe `Behaviour` (CAIRE, 2016).

Um agente pode ter vários comportamentos, os quais são executados cooperativamente. Quando um deles bloqueia, ele sai da fila de ativos e vai para a fila de bloqueados. Após a execução do método `action()` de um comportamento, o agente chama o método `done()`, o qual retorna `false` indicando que o comportamento ainda não terminou sua execução, ou `true` se o comportamento encerrou a execução, fazendo com que o Jade jamais volte a executá-lo sem um `reset` explicito.

Tendo como cenário um ambiente `JADE`, primeiramente acontece a criação do agente no `JADE`, que é uma instância da classe `Agent` ou de suas subclasses. Os desenvolvedores irão criar seus próprios agentes estendendo de `Agent`. Como classe filha o agente herda um conjunto de métodos básicos e interações com a plataforma como registro e configuração. Depois, quando o agente é executado, o método *done()* é inspecionado e enquanto retornar *False*, o comportamento é mantido em uma das listas de comportamentos, quando retornar *True* o comportamento é removido das listas de comportamentos do agente.

JADE define alguns comportamentos padrões que estão disponíveis e têm funcionalidades próprias:

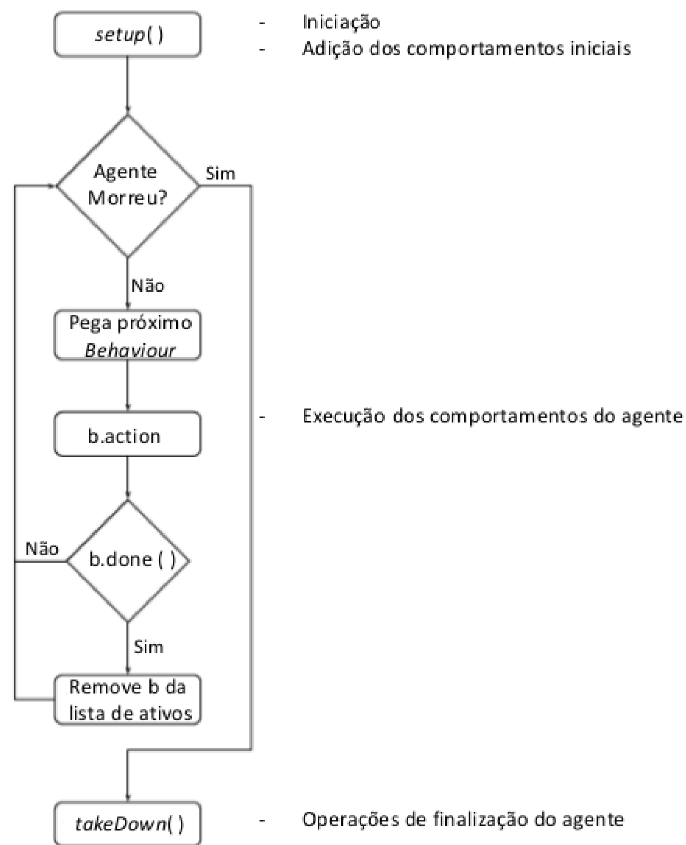


Figura 5 – Ciclo de Execução de um Agente. Fonte: (CAIRE, 2016)

- `OneShotBehaviour`: é a implementação da classe `SimpleBehaviour` com o método *done()* sempre retornando *True* fazendo com que o *action()* seja executado apenas uma vez;
- `CyclicBehaviour`: é a implementação da classe `SimpleBehaviour` com o método *done()* sempre retornando *False* fazendo com que o comportamento nunca termine;
- `WakerBehaviour`: executa a ação *onWake()* depois de esperar um determinado período de tempo para efetivamente executar a tarefa;
- `TickerBehaviour`: executa a ação *onTick()* periodicamente em um intervalo de tempo definido. Este comportamento nunca acaba, pois seu método *done()* sempre retorna *False*;
- `SequentialBehaviour`: agenda no Jade a execução de subcomportamentos de modo sequencial e termina quando todos são executados;
- `ParallelBehaviour`: agenda no Jade a execução de subcomportamentos de modo paralelo e pode terminar depois que todos ou de uma determinada quantidade de comportamentos terminarem;

- `FSMBehaviour`: agenda no Jade a execução de subcomportamentos como uma máquina de estados em que os eventos são os valores de retorno do método *onEnd()* e cada subcomportamento é um estado da máquina.

Jade permite o desenvolvimento de sistemas multiagentes e fornece serviços transparentes para a criação e instanciação de agentes, comunicação, registro e descoberta de serviços e administração da plataforma e seus contêineres.

2.3 Paradigmas

Paradigma é um modelo ou padrão a ser seguido. Pode ser um modelo físico ou mental, uma visão geral ou um modelo de referência. Eles podem ser aplicados em sistemas de produção com o objetivo de melhorarias em seus processos produtivos visando a gerar uma grande quantidade de produtos em um curto período de tempo.

Para os sistemas produtivos que, além dos requisitos tradicionais, atualmente, têm que lidar com os desafios da customização em massa, diversos paradigmas têm surgido para solucionar estes problemas. O primeiro aqui considerado é o *Flexible Manufacturing Systems* (FMS) que apresenta o início da flexibilidade na manufatura.

O conceito de manufatura flexível está associado a habilidade de se produzir diferentes produtos, com eficiência, através da reconfiguração dos recursos (SETHI, 1990). Essa abordagem, permite aumento do nível de utilização dos equipamentos, melhor qualidade de produto, redução do custo de produção e dos tempos de operação da máquina e o rastreio dos produtos ao longo da linha de produção (PEIXOTO, 2012).

Além de FMS, outros paradigmas têm surgido, tais como *Bionic Manufacturing Systems* (BMS), *Holonic Manufacturing Systems* (HMS), *Reconfigurable Manufacturing Systems* (RMS) e *Evolvable Assembly/Production Systems* (EAS/EPS), os quais serão rapidamente abordados abaixo.

2.3.1 *Bionic Manufacturing Systems* (BMS)

Usa conceitos de manufatura modular, inspirado no comportamento de sistemas biológicos. Foi originalmente sugerido por Ueda (1992), que observou as características de adaptação de auto-organização de seres vivos e a possibilidade de implementar tais características em sistemas industriais.

Nesse paradigma, o sistema de manufatura é comparado a um conjunto de células e órgãos que interagem dinamicamente entre si trocando informações, o que possibilita ao sistema reagir a mudanças que ocorrem no ambiente. Essas interações são alcançadas através da troca de informações e de ajustes entre os módulos do sistema.

O trabalho de Ueda (2007) mostra basicamente dois tipos de mensagens que trafegam no sistema, tipo DNA e tipo RNA. Essas mensagem são interpretadas pelos recursos do sistema, denominados de BN, que são os componentes do sistema (robô, garra, esteira e outros).

O DNA contém informações de como formar o corpo, ou seja, tem informações codificadas de "como fazer as coisas". O RNA é o trasporte, sendo que ele é responsável pelo deslocamento do DNA de um lado para outro. Assim, o que é programado no BN, por exemplo, é somente a maneira de ler o DNA, de forma que, cada componente do sistema de produção somente execute uma determinada ação limitada à informação contida no DNA, e o RNA é encarregado somente do trasporte.

O termo *Bionic*, com decorrer do tempo, passou a ser substituído por *Biological* por causa da comparação da estrutura biológica com a estrutura da manufatura.

2.3.2 *Holonic Manufacturing Systems* (HMS)

O termo holônico é inspirado na palavra *holon*, combinação de *holo*, que significa "o todo", e *on* que significa "parte"; ou seja, um *holon* é a parte de um todo ou o todo em uma parte.

Este conceito é muito usado atualmente. Um conjunto de *holons* forma uma hierarquia, chamada de *holonarquia*, em que os *holons* são hierarquicamente divididos; cada *holon* pode ser composto por um conjunto de outros *holons* menores ou fazer parte de um organismo (*holon*) maior simultaneamente.

HMS tem como princípio de que um sistema de manufatura é formado por diversas partes, que são, naturalmente, sistemas complexos *per si*. Cada um destes subsistemas são criados a partir de sistemas mais simples.

Brussel et al. (1998) criou sua implementação, hoje de referência, que utiliza conceitos de orientação a objetos, aplicados à produção, trabalho este, com o acrônimo PROSA, no qual são explicitados três tipos de *holons* básicos para o sistema:

- *Order holon* - emite ordens e inicia o processo de produção;
- *Product holon* - aguarda a mensagem *inicio* do *order holon* e detêm as informações do processo;
- *Resource holon* - recebe informações do *product holon* de quando deve atuar sobre o produto.

Os diferentes tipos de *holon* trocam mensagem entre si formando um ciclo e essas interações convergem para execução de determinado serviço, de forma que, a aplicação

do conceito holônico na manufatura resulta em um sistema hierárquico, modular, e mais flexível. Essa hierarquização e modularização fazem com que o sistema execute as suas funções através de cooperação de *holons*, e isto é chamado de comportamento holônico.

2.3.3 *Reconfigurable Manufacturing Systems* (RMS)

Um problema que aparece na produção em massa é quando há necessidade de realizar uma alteração qualquer na linha de produção, o que requer que o sistema seja reconfigurado, o que toma tempo e esforço. Assim, o ideal é que ele faça a própria reconfiguração.

Em um sistema de manufatura reconfigurável, existe um planejamento para garantir a capacidade de uma rápida reconfiguração e integração dos componentes através das características de modularidade, integrabilidade e customização (MEHRABI; ULSOY; KOREN, 1999).

Para atender às necessidade que podem ocorrer na produção, a capacidade de modificar o *software* e o *hardware* é importante (ELMARAGHY, 2006). Para isso, faz-se necessário o uso de máquinas inteligentes que possam modificar o seu *software* e/ou o seu *hardware*.

Por exemplo, em uma linha de montagem com máquinas inteligentes, a capacidade de modificação requer que o sistema tenha um banco de dados com todos as possibilidades de produtos. Esse banco de dados é feito pelo desenvolvedor que cadastra os novos produto no sistema. Quando o sistema começa a fabricar um produto novo, ele faz uma consulta no banco de dados, e este fornece o programa e as ferramentas necessárias à fabricação daquele produto, ou seja, caracterizando uma parte física e outra lógica como apresentado na Figura 6.

As características de flexibilidade, modularidade e reconfiguração fazem com que o sistema gaste o mínimo de tempo possível para modificar-se devido à entrada de um novo produto sem afetar muito a produção, isto porque um novo produto pode ser previamente planejado, de forma que, quando entrar em produção, apenas seja necessário o *download* do *software* e pequenos ajustes do *hardware* na linha, o que torna o sistema reconfigurável.

2.3.4 *Evolvable Assembly System* (EAS) e *Evolvable Production Systems* (EPS)

Os paradigmas EAS/EPS, de acordo com Onori (2002), compartilham características particulares com os paradigmas BMS, HMS e RMS como: modularidade, reconfiguração e adaptação, presentes em sistemas de montagem e sistemas produtivos (ALSTERMAN; ONORI, 2005).

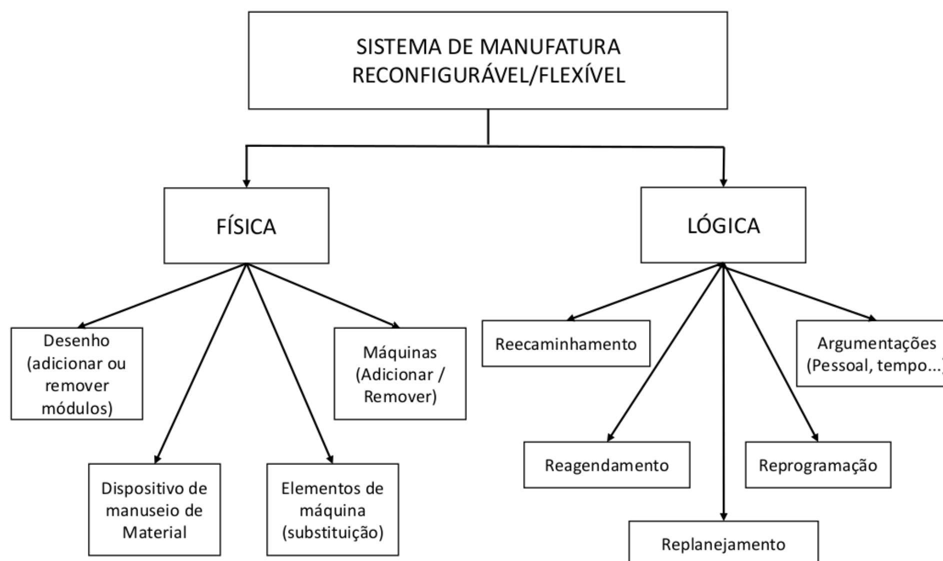


Figura 6 – Sistema RMS. Fonte: (ELMARAGHY, 2006), com adaptações.

O EAS visa ao desenvolvimento de sistemas de montagem evolutivo que são dinâmicos, auto-organizados e modularizados, de forma que sejam capazes de modificar sua estrutura de acordo com mudanças ambientais relevantes.

O EPS é definido como um sistema de produção evolutivo que, semelhante ao EAS, é dinâmico, auto-organizado e modularizado; entretanto possui a capacidade de fazer adaptações no processo, tais como inserção, troca ou remoção de componentes, enquanto o sistema está operacional (ONORI; BARATA, 2010).

Um sistema baseado no paradigma EPS tem duas características principais: evolução e adaptação. A adaptação ocorre quando o sistema, frente a perturbações externa, tem a capacidade de se ajustar para continuar operando conforme as especificações, por exemplo, quando o sistema consegue modificar certos parâmetros a fim de manter a produção. A evolução ocorre quando há uma alteração ambiental relevante, isto é, o sistema é capaz de modificar a sua estrutura para responder àquela mudança, por exemplo, configurando automaticamente novos módulos que entram e saem do sistema (RIBEIRO et al., 2010).

Na proposta deste trabalho, os módulos da linha de produção e os equipamentos de montagem têm gerenciamento próprio e são capazes de interagir, executar serviços e se reconfigurar, ou seja, evoluir de acordo com uma determinada situação ou necessidade.

Um sistema que externa suas funcionalidades apresenta capacidade de adaptar-se a flutuações de demanda e é capaz de evoluir conforme a entrada de novos produtos (os possíveis de fabricação com os módulos disponíveis ou módulos que podem ser incorporados ao sistema), não necessariamente planejados concomitantemente com o sistema produtivo, é chamado de sistema evolutivo (EPS) e alvo da proposta aqui apresentada.

2.4 Emergência e Auto-organização

A auto-organização e a emergência são fenômenos que podem ser observados em sistemas naturais como em um bando de pássaros migratórios, em um cardume de peixes ou ainda em cristais que crescem de forma organizada.

A emergência é descrita como um fenômeno em que o comportamento global surge a partir de interações locais do sistema. Por exemplo, em um formigueiro, o seu funcionamento é totalmente organizado e hierarquizado. Entretanto, o caminho de feromônio que as formigas formam é um comportamento complexo (caminho global), e este surge através de ações locais, resultado de simples interações de uma formiga com outra ou com o seu ambiente. Tal caminho é ótimo ou *quasi*-ótimo e surge por emergência. Estudando tal comportamento, Dorigo e Stutzle (2004) propõem o uso do conceito de emergência na implementação de um algoritmo de otimização baseado nas colônias de formigas.

Algumas características importantes são elencadas com relação ao conceito de emergência:

- efeito microestado ou efeito microestado - macroestado - as interações que acontecem localmente resultarão no comportamento global do macroestado. O macroestado refere-se a propriedades, comportamentos, estruturas ou padrões;
- novidade radical - emergentes, para sistemas complexos, emergentes, ou padrões do macroestado, não são vistos antecipadamente, ou melhor, não são completamente previsíveis no nível microestado;
- interação entre partes - as partes precisam interagir para que, por emergência, surja o comportamento no nível macro;
- controle descentralizado - sistemas complexos em que há emergência, não há um controle central que define o comportamento macro, isto é, somente ações e controles locais definem, por meio de interações, o comportamento macro(emergente).

Uma definição de auto-organização é dada por Wolf e Holvoet (2005) é "A auto-organização refere-se exatamente ao que é sugerido: sistemas que parecem organizar-se sem interferência externa, manipulação ou controle." Assim, um sistema é dito auto-organizado se ele adquirir uma estrutura espacial, temporal ou funcional de maneira que formem um sistema adaptável e dinâmico, em que adquiram e mantenham sua estrutura sem controle externo.

Com relação a auto-organização, serão elencadas algumas características importantes:

- autonomia - o sistema precisa auto-organizar-se sem interferência externa;
- dinâmica - é uma propriedade essencial no processo, pois ao longo do tempo, há um aumento da ordem do sistema, isto é, uma mudança para uma ordem maior;
- adaptabilidade ou robustez - em sistemas ditos auto-organizados, a robustez é usada como adaptabilidade frente a perturbações e mudanças, sendo esperado que este lide com essa mudança e tenha a capacidade de manter a sua organização de forma autônoma.

Os autores, contudo, fazem uma diferenciação com relação aos conceitos de emergência e de auto-organização. Para eles, "a emergência enfatiza a presença de uma novidade coerente de emergentes no nível macro do sistema (propriedade, comportamento, estrutura, padrões,...), como resultado das interações entre as partes no nível micro", já a auto-organização "enfatiza o aumento da dinâmica e da adaptação na ordem ou na estrutura sem o controle externo".

Esses conceitos, com frequência, são aplicados em sistemas mecatrônicos a fim de que estes, dado um evento que venha a ocorrer, possam adaptar-se às mudanças de condições simplesmente ajustando o processo e sua configuração. Assim, combinando os conceitos, concluímos que, considerando um sistema complexo, a auto-organização é a causa e a emergência é o resultado do processo de auto-organização, ou mais especificamente "as interações entre as entidades individuais são a auto-organização e esta situa-se no nível micro do processo emergente" (WOLF; HOLVOET, 2005).

2.5 Definições Básicas

Para melhor visualizar alguns conceitos e características que serão usados na proposta deste trabalho será apresentada algumas definições.

2.5.1 *Skill*

Uma funcionalidade ou serviço que o agente executa é chamado de *skill*, por denotar uma habilidade que o agente tem no sistema. Um *skill*, tal como uma chamada de método remoto, é definido por um nome, zero ou mais argumentos e seus tipos, zero ou mais propriedades e seus valores, e o tipo de retorno dessa funcionalidade.

As informações das *skills* dos agentes são disponibilizadas no sistema com o objetivo de oferecer esses serviços aos demais agentes. Podemos dizer, então, que uma *skill* é uma habilidade que o agente dispõe e que deve seguir um determinado padrão com o objetivo de facilitar sua consulta e chamada dentro do sistema.

Associar uma *skill* a um agente é vincular uma habilidade individual a um *Agent Idenfication* (AID) específico. Então, cada agente expõe *skills* para o sistema e cada *skill* pode está associado a um ou vários agentes. As arquiteturas EPS existentes partem do mesmo princípio e, algumas, expõe as *skills* na forma de contratos assemelhando-se a SOA, que é uma proposta orientada a serviços.

Os agentes se comunicam por meio mensagens e são endereçados em rede, de forma que, mesmo em máquinas diferentes, consigam completar a comunicação e executar os serviços.

Os serviços inerentes a cada agente podem ser executados local ou remotamente. A chamada para execução de serviço local é um procedimento nativo, pois a *skill* está no próprio agente. Quando essa *skill* não pertence ao agente, é realizada uma chamada de execução de serviço remota. Dentro da arquitetura EPS, as *skills* têm assinaturas específicas, o que as define, e são registradas e disponibilizadas no sistema.

2.5.2 Máquina de Estados

Uma máquina de estados tradicional ou máquina de estados finitos (*FSM - Finite State Machine*) é um modelo matemático usado para representar programas de computadores ou circuitos lógicos. O conceito é concebido como uma máquina abstrata que deve estar em um de seus finitos estados. A máquina está em apenas um estado por vez, este estado é chamado de estado atual. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada num estado, no início do sistema, até o momento presente. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento (GOH MOHAN BARUWAL CHHETRI, 2007).

Pode-se resumir as partes básicas da máquina de estados como:

- *State*: um estado representa a situação em que um objeto se encontra em um determinado momento durante o período que este participa de um processo. Um objeto pode passar por diversos estados dentro de um mesmo processo. Um estado pode demonstrar:
  - a espera pela ocorrência de um evento;
  - a reação a um estímulo;
  - a execução de uma atividade;
  - a satisfação de alguma condição.

- *Transition*: representa um evento que liga dois estados, isto é, executar uma transição é executar ações que realizam a mudança de um estado em outro no sistema;
- *Action*: representa a ação que será tomada dado o surgimento de um evento pré-definido;
- *Event*: é uma alteração externa mapeada na FMS e que impacta na sua evolução.

2.5.3 XML

XML é a sigla para Extensible Markup Language ou Linguagem Extensível de Marcação Genérica. É recomendada para gerar linguagens de marcação para necessidades especiais tais como: criação de documentos com dados organizados hierarquicamente, textos, banco de dados ou desenhos vetoriais. XML é capaz de descrever diversos tipos de dados, e seu objetivo principal é a facilidade de compartilhamento de informações.

O termo "Linguagem de marcação" é um agregado de códigos que podem ser aplicados a dados ou textos para serem lidos por computadores ou pessoas. O XML traz uma sintaxe básica que pode ser utilizada para compartilhar informações entre diferentes computadores e aplicações. Portanto, uma das suas principais características do XML é sua portabilidade, pois, um banco de dados pode, por exemplo, escrever um arquivo XML para que outro banco consiga lê-lo. Como benefícios para desenvolvedores e usuários temos:

- buscas mais eficientes: os dados em XML podem ser unicamente "etiquetados", o que permite que, por exemplo, uma busca por livros seja feita em função do nome do autor;
- desenvolvimento de aplicações flexíveis para a Web: os dados XML podem ser distribuídos para as aplicações, objetos ou servidores intermediários para processamento;
- integração de dados de fontes diferentes: O XML permite os dados possam ser facilmente combinados via software em um servidor intermediário, estando os bancos de dados na extremidade da rede;
- computação e manipulação locais: os dados XML recebidos por um cliente são analisados e podem ser editados e manipulados de acordo com o interesse do usuário;
- atualizações granulares dos documentos: somente os elementos modificados seriam enviados pelo servidor para o cliente.

Alguns dos propósitos do XML são: auxiliar os sistemas de informação no compartilhamento de dados (especialmente via internet), codificar documentos e inserir seriais nos

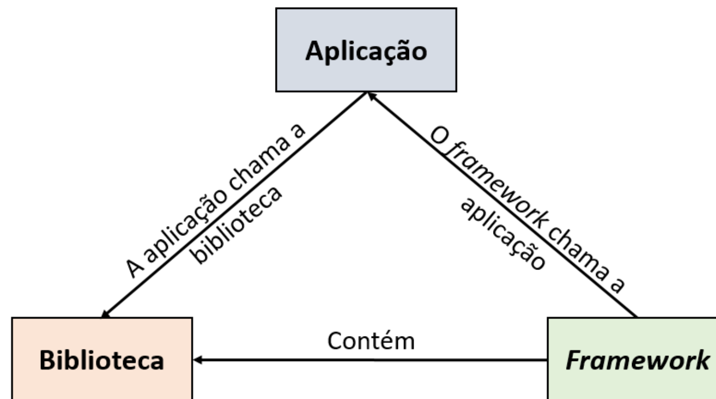


Figura 7 – Relação entre *framework*, Aplicação e Bibliotecas. Com adaptações de (Wikipedia, 2016)

dados comparando o texto com o de outras linguagens baseadas em serialização. Assim, neste trabalho, o uso da linguagem XML tem como objetivo definir um padrão simples, no qual seja possível decodificar as informações sem muito custo computacional.

2.5.4 `API`, Biblioteca e *Framework*

Uma *Application Programming Interface* (`API`) é um conceito abstrato que define formas de executar uma tarefa específica. É a maneira que o código se relaciona com uma biblioteca. Podemos dizer, também, que é a documentação que determina como um programador pode realizar uma tarefa através de uma biblioteca. Simplificando, a `API` é um conjunto de regras para realizar uma determinada tarefa.

Uma biblioteca é uma implementação real das regras de uma `API`. Portanto, ela é mais concreta. A biblioteca precisa respeitar as regras da `API` sempre, sendo que a biblioteca costuma ser autossuficiente. Dessa forma, podemos concluir que uma biblioteca é uma ferramenta na qual se usa somente o que se precisa.

Um *framework* é um conjunto de bibliotecas usadas para conseguir executar uma operação maior. É comum um *framework* encapsular os comportamentos da `API` em implementações mais complexas, permitindo o seu uso de forma mais flexível, frequentemente através de extensões e configurações. Pode ser considerada uma camada em cima da `API`.

Conforme mostrado na Figura 7, um *framework* é uma arcabouço para desenvolvimento de aplicações e, portanto, o desenvolvedor completa as partes faltantes daquele *framework*. Este se utiliza de uma ou mais bibliotecas para realizar o seu serviço; e as aplicações, igualmente, fazem uso de bibliotecas, mesmo que conjuntamente a *frameworks*.

A arquitetura proposta, chamada de EPSCore, é um *framework* para o desenvolvimento de aplicações EPS, baseada no conceito de agentes mecatrônicos e chamadas

de *skills*. Um exemplo de desenvolvimento de uma aplicação, pode ser encontrado na aplicação SIAPE.

2.6 Simulação de Sistemas

Um simulador é uma máquina que reproduz o comportamento de um sistema sob determinadas condições, permitindo uma maior aproximação com a realidade. Os simuladores costumam combinar partes mecânicas e/ou eletrônicas e partes virtuais que ajudam a simular a realidade e podem ser usados no âmbito profissional, como instrumento de treinamento, como avaliação de eficiência ou funcionamento em um sistema, prevenção de comportamentos não desejados em máquinas, entre outras funcionalidades. A simulação é uma forma de imitar a realidade sem correr os riscos, os custos e o tempo que resultariam se tivéssemos de experimentar.

Existem diversas definições para a simulação, dentre elas podemos citar:

- Pegden, Sadowski e Shannon (1995): "a simulação é um processo de projetar um modelo computacional de um sistema real e conduzir experimentos com este modelo com o propósito de entender seu comportamento e/ou avaliar estratégias para sua operação". Sendo interpretado como processo que engloba a construção do modelo, descrição do comportamento do sistema, uso do modelo para prever o comportamento futuro, isto é, os efeitos produzidos por alterações no sistema ou nos métodos empregados em sua operação;

- Schriber (1974): "simulação implica a modelagem de um processo ou sistema, de tal forma que o modelo imite as respostas do sistema real em uma sucessão de eventos que ocorrem ao longo do tempo";

- Kelton e Law (2000): "a simulação como uma técnica que utiliza computadores para imitar as operações de vários tipos de processos e facilidades do mundo real";

- Banks (2000): "a simulação é a imitação da operação de um processo ou sistema do mundo real ao longo do tempo";

- Kelton, Sadowski e Sadowski (2002): "simulação é o processo de projetar e criar um modelo em um computador de um sistema real ou proposto para o propósito de conduzir experimentos numéricos para nos dar uma melhor compreensão do comportamento de um dado sistema dada uma série de condições".

Um bom exemplo de simulação é aquele usado na indústria aeronáutica, em que a aerodinâmica dos aviões, em projeto, é testada em túneis de vento através de pequenas maquetes que apresentam o mesmo formato do avião, ou seja, é o "modelo" do avião

real. Como seria completamente inviável construir todo o avião e tentar fazê-lo voar com pilotos de prova, é necessário o uso da simulação que evita, nesse caso, a perda de vidas e investimentos.

Um piloto usará um simulador para experimentar sensações físicas similares àquelas que sentiria em pleno voo. Isso será possível, pois o computador é alimentado com as propriedades e características de sistemas reais, criando um ambiente “virtual”, que é usado para testar as teorias desejadas. O computador efetua os cálculos necessários à interação do ambiente virtual com o objeto em estudo e apresenta os resultados do experimento na forma de gráficos e relatórios.

De acordo com Pegden, Sadowski e Shannon (1995), o processo de simulação é amplo e engloba não apenas a construção do modelo, mas todo o método experimental que busca:

- descrever o comportamento do sistema;
- construir teorias e hipóteses considerando as observações efetuadas;
- usar o modelo para prever o comportamento futuro, isto é, os efeitos produzidos por alterações no sistema ou nos métodos empregados em sua operação.

Algumas características são alcançadas através do uso de simulação:

- previsão do comportamento do sistema;
- um dispositivo para compreensão de um problema;
- uma ferramenta de análise para ajustes no funcionamento;
- economia de recursos financeiros;
- segurança na execução dos testes;
- melhorias relacionadas à eficiência e à robustez;
- uma ferramenta de projeto para avaliar problemas e propor soluções;
- visualização do funcionamento do processo;
- uma ferramenta de treinamento.

3 Estado da Arte

3.1 *Multi Agents System* (MAS) e *Service Oriented Architecture* (SOA)

Os sistemas multiagentes (MAS) e a arquitetura orientada a serviços (SOA) surgem como propostas para solucionar desafios relacionados à agilidade e flexibilidade na manufatura.

Multi Agents System (MAS) é capaz de modelar e implementar comportamentos individuais e sociais em um sistema distribuído. Usa agentes para oferecer serviços como registro, negociação e alocação de recursos. Esse sistema é formado por uma coalizão de agentes e o resultado é um sistema com capacidade de reconfiguração e auto-organização.

Em um sistema multiagente os agentes são entidades que executarão determinado serviço e interagirão com o ambiente e uns com os outros. Assim, a cooperação forma uma rede distribuída que tem a capacidade de resolver coletivamente os problemas, sendo que, para realizar a implementação desse sistema, é necessária a existência de uma plataforma de agentes no qual a arquitetura destes é instanciada. Os agentes são criados de forma independente, operam independentemente do meio (flexibilidade), fornecem funcionalidades próprias (autossuficiência) e podem funcionar sozinhos (autônomos).

Arquitetura Orientada a Serviços (SOA) são sistemas interoperáveis onde as funcionalidades implementadas são disponibilizadas na forma de serviços. Em sequência, cada sistema dentro do SOA fornece serviços independente dos demais e quando conectados, disponibilizam esses serviços a outros do sistema.

SOA usa o conceito de serviços e MAS usa o conceito de agentes. Assim podemos inferir que com um MAS é possível fazer um SOA, mas não dá para fazer um Sistema Multiagentes com SOA. As principais características e diferenças entre esses sistemas são apresentadas por Ribeiro, Barata e Mendes (2008) na Tabela 2.

3.1.1 *Evolutionary MAS* (EMAS)

Seguindo o mesmo raciocínio com relação ao uso de agentes, recentemente, Byrski et al. (2015) propôs o desenvolvimento de uma arquitetura denominada *Evolutionary Multi-Agent Systems* (EMAS) que é uma proposta para resolver globalmente problemas de otimização no sistemas de manufatura.

Na proposta da arquitetura EMAS (veja Figura 8), os processos evolutivos são naturalmente descentralizados e os agentes são capazes de se reproduzir e morrer de acordo

Tabela 2 – Comparativo de MAS e SOA

	Características	Resumo
SOA	Autonomia Interoperabilidade Encapsulamento Disponibilidade Web Services	É um sistema que não tem dependência direta, tem capacidade de cooperação e integração, assim como autoadaptação, serviçãoos web em tempo real e gerenciamento de serviços, que podem ser publicados e disponibilizados. Os serviços, a inteligência e sensoriamento são encapsulados de forma a reduzir a complexibilidade.
MAS	Autonomia Sociabilidade Racionalidade Reatividade Proatividade Adaptabilidade	Um agente deve ser capaz de se comunicar com outros agentes, raciocinar com relação aos dados que recebe, reagir ao ambiente se adaptando à situação que venha a ocorrer e aprender sempre buscando o ótimo, ou seja, principios de auto-otimização e autoadaptação

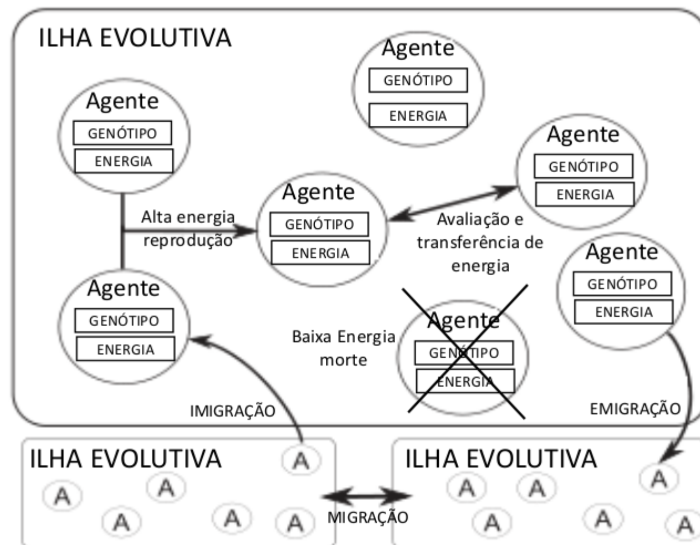


Figura 8 – `EMAS` - Sistema Multiagentes Evolutivo. Fonte: (BYRSKI et al., 2015), com adaptações

com a necessidade do sistema. O modelo básico mostra onde são definidos os agentes e os recursos (chamado energia). Por exemplo, quando um agente encontra um vizinho (que tem uma quantidade menor de energia), este passa parte de sua energia para o vizinho, sendo que esta ação pode acontecer das seguintes formas:

- reprodução - quando o nível de energia é elevado a um determinado patamar, é iniciada a produção de um novo indivíduo em cooperação com um de seus vizinhos;
- morte - o agente é removido do sistema, quando a energia cai para um determinado nível, sua energia restante é dividida entre os vizinhos;

- migração - quando o nível de energia se eleva a um determinado patamar, ocorre a remoção do agente e sua transferência para uma topologia pré-definida.

Nessa arquitetura, cada uma das ações possíveis tem uma determinada probabilidade de acontecer, ou seja, é uma escolha aleatória e é realizada apenas quando suas pré-condições básicas são satisfeitas.

As abordagens propostas pelas arquiteturas apresentadas propõem basicamente o funcionamento do sistema como uma equipe, no que todos os elementos trabalham para atingir um mesmo objetivo, sendo que esses elementos têm as mesmas informações relativas ao sistema e possuem o mesmo direito quando da tomada de alguma decisão. Porém, cada um deles deve saber qual o seu papel, de maneira que, no momento em que surge um problema, o elemento mais indicado seja alocado para resolvê-lo e o sistema não se torne anárquico.

Igualmente, a abordagem adotada neste trabalho também é o uso de agentes inteligentes, mais especificamente, agentes mecatrônicos interagindo na forma de sistema multiagentes para proporcionar auto-organização e emergência.

Enfim, a finalidade é melhorar, organizar e otimizar os processos na manufatura. A escolha de qual padrão, arquitetura ou caminho se deseja seguir depende da aplicação e dos resultados que se deseja alcançar.

3.2 Ontologia

O conceito de ontologia é definido por Chandrasekaran, Josephson e Benjamins (1999) como: "(...) a especificação de uma conceituação", ou seja, se refere a uma definição formal de um dado conhecimento que descreve algum domínio, normalmente um domínio conhecido por todos os agentes da aplicação, usando um vocábulo de representação e pode ser representado através do universo de discurso, que indica um conjunto de vocábulos relevantes. A conceituação refere-se à abstração de uma parte do mundo (o domínio) onde são representados os conceitos relevantes e seus relacionamentos.

Tamma et al. (2002) diz que o uso da ontologia é como um objetivo complexo que é dividido em subobjetivos que são usados para descrever a transição de estados a partir de um estado inicial (pré-condição) e um estado final (pós-condição).

Usamos ontologias para ajudar na significação das coisas, mais precisamente no entendimento. A comunicação é realizada por meio da linguagem (gesto, fala, sinais). Essa linguagem, por exemplo, a fala, refere-se a um conjunto de palavras, frases e vocábulos e na comunicação é importante que seja padronizada e que todos os participantes da conversa falem a mesma língua.

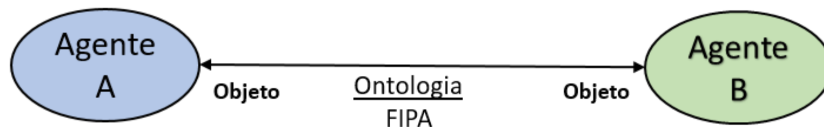


Figura 9 – Comunicação entre agentes usando ontologia.

Falar a mesma língua é quando o significado dos vocábulos são os mesmos. Dada a comunicação entre "A" e "B", por exemplo, a representação do conhecimento e o conhecimento em si de algo devem ser iguais para que seja possível o entendimento da informação.

Ao usar ontologia é importante definir uma estrutura comum de significados para serem usados pelos agentes na comunicação sem que aconteça a possibilidade de ambiguidades. Assim, é criado um vocabulário e semântica próprios para a troca de mensagens.

Uma ontologia define, portanto, o que se entende sobre os conceitos do mundo que são representados dentro do sistema por meio de suas classes e objetivos e que são entendidos por todos os agentes do sistema multiagentes.

A construção de ontologias envolve a definição de um domínio e um escopo e, posteriormente, são escolhidas uma metodologia, uma ferramenta e uma linguagem para sua especificação.

Na figura 9 os agentes estão comunicando entre si e consultando a ontologia. Ao comunicar-se, os agentes usam a ontologia para padronizar a comunicação. A comunicação entre os agentes "A" e "B". Pode ser realizada, por exemplo, por meio do padrão FIPA, através de mensagens ACL que contém ações e conceitos da ontologia utilizada.

FIPA oferece um campo, dentro de uma mensagem ACL, específica para descrever os elementos que os agentes podem usar dentro do conteúdo das mensagens, isto é, sua ontologia. Ela define um vocabulário e as relações entre os elementos de tal vocabulário. As relações podem ser estruturais, por exemplo, um predicado PAI pode aceitar dois parâmetros, o nome de um Pai e um conjunto de Crianças, ou semânticas, por exemplo, o conceito da classe Homem também é o da classe de Pessoa.

Definir ontologia no `JADE` significa implementar a comunicação entre os agentes diferenciando pelo tipo de conteúdo:

- Tipo *String*: é forma mais básica e consiste em utilizar *Strings* para representar o conteúdo das mensagens. Isto é útil quando o conteúdo das mensagens de dados é

atômica, mas não no caso de conceitos objetos abstratos ou dados estruturados. Em tais casos, a cadeia precisa ser analisada para acessar suas diversas partes;

- Tipo *Java Objects*: é uma estrutura que explora a tecnologia `Java` para transmitir objetos serializados Java diretamente como o conteúdo das mensagens. Este é frequentemente um método conveniente para uma aplicação local, onde todos os agentes são implementados em `Java`;
- Tipo *Ontology Objects*: envolve a definição dos objetos a ser transferido como uma extensão das classes predefinidas para que Jade pode codificar e decodificar mensagens em um formato padrão `FIPA`. Isso permite aos agentes de `JADE` interoperar com outros sistemas de agente.

A ontologia do JADE permite definir algumas mudanças em:

- *Predicates* - são expressões que dizem alguma coisa sobre o *status* do mundo e podem ser verdadeiras ou falsas;
- *Entities (or terms)* - são expressões que identificam entidades (abstratas ou concretas) que "existem" no mundo e que os agentes conversam e reagem com estas;
- *Concepts* - são expressões que indicam entidades com estruturas complexas que podem ser definidas em termos de *slot*. É uma interface genérica, em geral definida como referência dentro do predicado;
- *AgentActions* - é um conceito especial que indica ações que podem ser realizadas pelos agentes.

Existem diversas implementações de ontologia: *Resource Description Framework* (`RDF`) (CORBY O.; DIENG, 2000), *Web Ontology Language* (`OWL`) (HORROCKS I.; PATEL-SCHNEIDER, 2003), *Web Service Modeling Ontology* (`WSMO`) (FEIER et al., 2005; FENSEL et al., 2011) e *Semantic Markup for Web Services* (`OWL-S`)(BURSTEIN et al., 2004) que têm como principal objetivo fornecer e providenciar soluções em casos em que a informação está em documentos e é necessário o processamento e interpretação desta informação.

Este trabalho utiliza, no que diz respeito à padronização da linguagem, o conceito de ontologia com a finalidade de facilitar e padronizar as informações trocadas entre os agentes.

3.3 EPS

Desde os tempos de Ueda, Vaario e Fujii (1998), que desenvolveram uma estrutura para sistemas multiagentes, nos quais havia a proposta de simulação de sistemas de

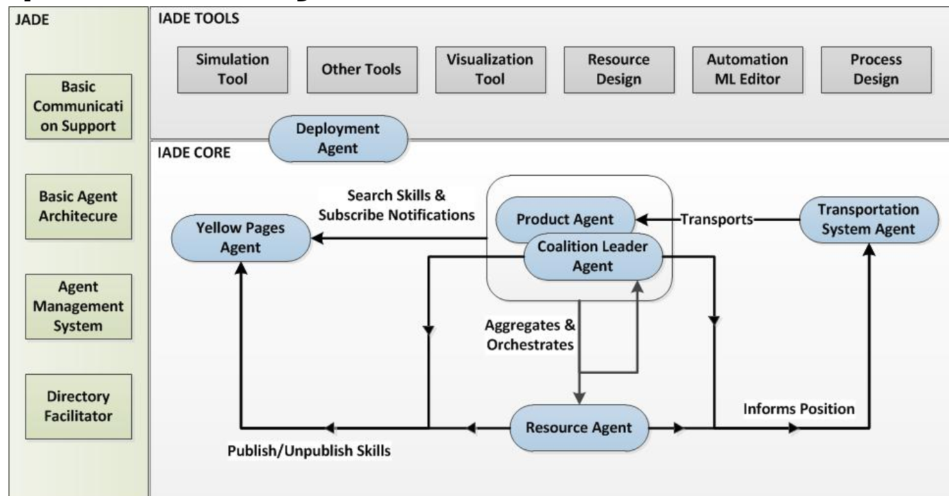


Figura 10 – IADE (IDEAS Agent Development Environment). Fonte: (ONORI et al., 2012)

produção baseados em sistemas naturais, que a proposta holônica visa a capacidades como auto-organização, autoadaptação e autorreparação. Outros trabalhos interessantes são (LEITAO; RESTIVO, 2006; VALCKENAERS; BRUSSEL, 2005) que têm desenvolvido esse conceito e, ao aplicar essa estrutura, foi possível reduzir a complexidade e permitir a reconfiguração, como os trabalhos de BARBOSA et al. (2011, 2015), que propõem a auto-organização a partir de visões micro e macro do sistema.

A arquitetura EPSCore é uma das propostas deste trabalho e é representado por estrutura simples, parte de uma plataforma, que auxilia no aprendizado de inteligência artificial distribuída através do uso do conceito de agentes inteligentes aplicados à manufatura. Ela é fundamentada basicamente no conceito de sistemas multiagentes, EPS e agentes mecatrônicos. De forma que, uma característica primordial é que o foco da inteligência do processo produtivo, isto é, o conhecimento de como se faz um determinado produto, é retirado dos módulos(módulos mecatrônicos) que realizam a atividade de montagem. A confecção desse produto acontece depois de interações entre os agentes do sistema.

A arquitetura EPSCore foi inspirada na proposta do projeto IDEAS (*Instantly Deployable Evolvable Assembly System*) desenvolvido na União Europeia (ONORI et al., 2012). O núcleo deste projeto é a uma plataforma de desenvolvimento de sistemas multiagentes para a manufatura, baseada no conceito de agentes mecatrônicos, chamada de IADE *(IDEAS Agent Development Environment)*. A Figura 10 mostra a *framework* da IADE com as ferramentas e a arquitetura. Abaixo são listados os principais agentes da arquitetura IADE:

- RAs (*Resource Agents*): são os agentes que externam serviços ao sistema de montagem;

- CLAs (*Coalition Leader Agents*): são coordenadores e organizam outros agentes mecatrônicos;
- PAs (*Product Agents*) : são os CLAs de alto nível que executam o processo de montagem do produto;
- TSAs (*Trasportation System Agents*): são abstrações do sistema de transporte, assim, apesar dos vários tipos de transporte existentes, estarão todos sob uma mesma interface.

Na arquitetura IADE existe um agente especial chamado de DA(*Deployment Agent*) que é responsável pela serialização das informações necessárias ao sistema e dinamicamente instancia os agentes mecatrônicos com todas as suas *skills*. A existência de uma forma serializada para os agentes mecatrônicos torna possível a instanciação sob demanda na máquina local, isto é, quando a ação desejada será executada. Assim, o sistema é capaz de instanciar dinamicamente, em módulos, agentes remotos.

O objetivo da proposta do desenvolvimento da arquitetura EPSCore é, assim como na IADE, disponibilizar uma infraestrutura de desenvolvimento de aplicações multiagente para a manufatura, isto é, que seja capaz de suportar agentes mecatrônicos.

Na literatura, destacam-se ainda alguns trabalhos que têm fundamentos no paradigma da manufatura EPS: Peixoto (2012) propõe uma arquitetura orientada a serviços para gerenciamento de sistemas de manufatura utilizando sistemas multiagente, Cavalcante (2012) e Rosa (2013) têm a base de seus trabalhos no projeto IDEAS(*Instanttly Deplyble Assembly System*) e esses trabalhos são:

- Cavalcante (2012): propõe uma arquitetura multiagentes híbrida, que é caracterizada pela capacidade do agente reagir a eventos externos, tendo como foco o conceito de auto-organização aplicada na manufatura;
  - Essa proposta é fundamentada em agentes mecatrônicos e voltada à abordagem do paradigma EPS. Os agente desenvolvidos são:
    - Agente Recurso (RA: *Resourse Agent*) - agentes que fornecem serviços no sistemas;
    - Agente Produto (PA: *Product Agent*) - agentes que solicitam serviços;
    - Agente de Transporte (TA: *Trasnport Agent*) - agente responsável pelo transporte dos agentes (PA) no sistema.
- Rosa (2013): propõe o uso dos conceitos de emergência e auto-organização através da sua implementação que dá suporte aos conceitos e interações entre os agentes, buscando aumentar a auto-organização e consciência do sistema mecatrônico.

As propostas acima citadas são derivadas também do IADE e ambos os trabalhos propõem melhorias para a plataforma e são focados em auto-organização e emergência. Assim, tendo esse princípio a arquitetura IADE foi a base de inspiração para desenvolvimento da arquitetura EPSCore e seus agentes têm somente a inteligência de seu próprio funcionamento e são implementados como agentes mecatrônicos, que possuem a capacidade de ação no mundo físico, através de seus sensores e atuadores, capacidade de computação própria e capacidade de comunicação com as outras entidades do sistema.

Dessa forma, nenhum dos agentes do sistema contém *toda* a informação necessária para se fabricar um determinado produto, todavia os produtos são montados em um ambiente dinâmico. A inteligência de como será montado um produto está no sistema, porém distribuída em todas as partes que o formam. Dessa forma, a integração das partes, ou melhor, a coalizão entre os agentes, é o que torna possível as atividades de montagem.

É importante mencionar também duas arquiteturas baseadas em `EAS/EPS`: *ADAptive holonic COntrol aRchitecture* (ADACOR) apresentada por Leitao e Restivo (2006), e a *Coalition Based Approach for Shop Flor Agility* (CoBASA) apresentada por Barata, Camarinha-Matos e Onori (2005).

A ADACOR é uma interessante arquitetura baseada no paradigma holônico mencionado na seção 2.3.2. Existe a possibilidade de esta arquitetura ser implementada em um sistema multiagente. Assim, o *holon* seria modelado como um agente e o sistema, como um todo, como um sistema multiagente. São apresentados os princípios fundamentais para sistemas de produção reconfiguráveis baseando-se na arquitetura ADACOR e incorporando mecanismos inspirados em outras áreas da ciência como a Biologia, a Teoria da complexidade e da vida artificial.

Na abordagem da arquitetura ADACOR, é vislumbrada a possibilidade de se desenvolver sistemas de controle de manufatura, com autonomia e capacidades inteligentes, ágeis e com rápida adaptação às mudanças do ambiente, preparados para lidar eficientemente com as ocorrências de distúrbios, e possibilitando a integração fácil entre recursos de manufatura e sistema.

A arquitetura CoBASA foi uma das primeiras a utilizar o conceito de agentes mecatrônicos e suas interações para modelar um sistema de montagem, a fim de desenvolver uma arquitetura genérica, baseada no conceito de EPS. Nesta proposta, agentes são agregados por meio de coalizões, por exemplo: coalizão 1, coalizão 2, e assim por diante, formado pelos seguintes agentes:

- MRA - *Mechatronic Resourse Agent* - é um recurso do sistema sendo formado pelo equipamento, o controle e o agente;

- AMI - *Agent Machine Interface* - é um agente que controla o módulo de *hardware*, representa a ligação entre o o equipamento mecatrônico e o agente;

- BA - *Broker Agent* - agente responsável pela criação das coalizões do sistema;

- CMgA - *Cluster Manager Agent* - responsável pelo gerenciamento de vários agentes;

- CA - *Coordination Agent* - agente responsável pela coordenação de outros agentes, ou seja, promove a coordenação entre os agentes, a fim de agregar os serviços relativos a cada agente.

Na arquitetura CoBASA, existe associação das habilidades e funcionalidades dos módulos do sistema de montagem, isto é, seus *skills*. Além disso especifica que um agente mecatrônico é a entidade que reconhece a chamada e a execução de seus *skills* no sistema de montagem.

Podemos citar também Frei et al. (2009) que propôs um sistema evolutivo que pode co-evoluir juntamente com o produto e o processo de montagem, de forma simples. Essa proposta é baseada em sistemas simples que permitam evolução contínua.

Outros pesquisadores como Onori, Barata e Frei (2006) e Ribeiro et al. (2010) estabeleceram preceitos que regem o paradigma EPS como, por exemplo, a necessidade de sistemas dinâmicos evoluírem para atender às mudanças de requisitos e sistemas que sejam baseados em autossuficiência, modularização e auto-organização. A formulação desses preceitos tem como objetivo atender a diferentes funcionalidades e, para isso, foi proposto um sistema que leva em consideração as seguintes características:

- modularização - módulos independentes que estão presentes em sistemas flexíveis;

- plugabilidade - a habilidade de lidar com a inserção de novos módulos enquanto o sistema está funcionando. O sistema precisa ser capaz de se reformular afim de manter sua eficiência;

- granularidade - um sistema tem que ter granularidade, que é a extensão à qual um sistema é dividido em partes pequenas, pode ser a representação de um sistema completo ou uma parte, uma descrição ou observação. "Ela é a extensão até à qual uma entidade grande é subdividida".

- reconfiguração - um sistema precisa ser capaz de se reconfigurar ou reformular seu *layout* sem prejudicar as outras funcionalidades ou serviços do sistema.

Na proposta deste trabalho, os módulos da linha de produção e os equipamentos de montagem têm gerenciamento próprio e são capazes de interagir, executar serviços e se reconfigurar, ou seja, evoluir de acordo com uma determinada situação ou necessidade.

Um sistema que mostra as funcionalidades apresentadas tem a capacidade adaptar-se à entrada de vários produtos, sendo que estes fazem parte de um faixa de produtos possíveis de fabricação no sistema.

Tabela comparativa dos paradigmas

A Tabela 3 apresenta o resumo dos paradigmas da manufatura, suas característica e os principais pesquisadores.

Tabela 3 – Resumo dos Paradigmas

Paradigmas	**Características**	**Principais Referências**
FMS	Todas as variacões do processo devem ser privistas.	SETHI,1990
BMS	Paralelo entre sistema biológico e o sistema de manufatura	UEDA,2007
HMS	Sistema holístico modularizado e hierarquizado	VALCKNAERS;BRUSSEL,2005
RMS	Máquinas reconfiguráveis	ELMARAGHY,2006
EAS/EPS	Sistema de agentes mecatrônicos	ONIRI,BARATA,2010

FSM é o paradigma inicial. Nele a alteração do produto implica a modificação da linha através da ajustes para iniciar sua produção, ou seja, é necessária a parada para a fabricação de um novo produto. Para reduzir essas paradas na hora da produção de um novo produto surge o RMS que tem como objetivo que o *hardware* e *software* possam ser reconfigurados de forma a atender a um requisito previsto anteriormente.

Com a evolução dos sistemas, surgem os paradigmas baseados em sistemas biológicos e holísticos com características como modularidade e hierarquização presentes nos paradigmas BMS e HMS.

Mais adiante, surge o paradigma EPS que traz o conceito de agentes e de módulos mecatrônicos. Este trabalho tem como base o uso deste paradigma sendo proposta uma implementação do paradigma EPS aplicada à manufatura.

3.4 Simulador

A simulação permite que se faça uma análise do sistema sem a necessidade de interferir no mesmo. Todas as mudanças e consequências, por mais profundas que sejam, ocorrerão apenas com o modelo computacional e não com o sistema real. Dessa forma, é possível melhorar o planejamento e a eficiência do sistema ao máximo fazendo uso desta ferramenta.

Um estudo de simulação consiste em várias etapas bem definidas, por exemplo, a formulação do problema conceitual, modelagem, entrada e análise de dados de saída, tradução do modelo/implementação, verificação, validação e experimentação (MUSTAFEE et al., 2015). Portanto, a simulação permite que se faça uma análise detalhada o tanto quanto possível de sistemas reais, sem a necessidade de interferir no mesmo.

Ribeiro, Barata e Ferreira (2010) expõem os requisitos para realização de diagnóstico no sistema e comenta que, uma vez que EPS é evolutivo e adaptável e tem capacidade de *plug-and-produce*, as abordagens de diagnóstico tradicionais dependem tipicamente de um modelo/descrição estática do sistema no qual o diagnóstico é realizado e, para diagnosticá-los com precisão, exigem o mapeamento cuidadoso de todas as relações. E completa que o objetivo do método usado é permitir que cada agente individual (módulo EPS) ajuste o seu estado interno utilizando apenas informação local, sendo considerada uma abordagem probabilística que ajuda na aprendizagem e de como o agente ganha mais experiência sobre o ambiente circundante.

Dessa forma, usando Simulação e EPS juntos, em um cenário fictício, para exemplificar uma aplicação industrial plausível: a situação de fabricação de automóveis em uma fábrica. Esta fábrica pretende construir uma linha de montagem para um modelo novo. O planejamento inicial dessa linha prevê o uso de um "braço" robótico para pintura do chassi. Ao usar o simulador, é possível, por exemplo, descobrir que adicionando mais um "braço" robótico aumente-se a eficiência do sistema como um todo e consegue-se uma produção maior em um tempo menor. Dada a situação fictícia, se não houvesse o uso do simulador, a "melhoria" ou o "problema" só seria verificado quando o sistema estivesse funcionando, e este teria que parar para ser realizada tal modificação.

Em um trabalho mais recente, Neves et al. (2014) focam em projetar e implementar uma ferramenta de software que simule o comportamento de auto-organização dos sistemas mecatrônicos levando em consideração o paradigma da manufatura EPS como base para desenvolvimento de sua ferramenta, que permite a criação do layout e da implantação de todos os agentes necessários ao sistema.

Rahatulain, Qureshi e Onori (2014) propõem o uso de um bloco no *Simulink* para simulação de eventos discretos, chamado de *SimEvents* com a finalidade de fazer análise de EPS relativas a diferentes aspectos do desenvolvimento: como especificações de requisitos, verificação, validação e geração de código. Neste trabalho, cada tipo de agente foi desenvolvido e testado separadamente para variar o número de entradas e saídas. Todos os agentes foram integrados e validados usando um protótipo industrial disponível na *Vinnova UDI Project*.

4 Proposta

Os sistemas de produção modernos têm de lidar com o problema tradicional de produção em massa, aumentando a qualidade dos produtos, ao mesmo tempo em que lidam com questões de mercado, que exigem ciclos mais curtos de vida dos produtos e tempos de produção reduzidos. Este cenário está presente em muitas fábricas onde os produtos devem ser fabricados para clientes seletos com exigências muito específicas. Assim, surgem propostas de arquiteturas que propõem soluções para a fabricação de produtos que devem ter alta variabilidade dentro de pequenos lotes de produção, a chamada customização em massa.

4.1 Proposta de Arquitetura

A arquitetura desenvolvida é denominada EPSCore. Ela é baseada no paradigma da manufatura EPS e é inspirada na arquitetura IADE. A implementação desta arquitetura é parte da proposta deste trabalho e leva em consideração uma visão do sistema de manufatura, em que cada elemento é modelado na forma de agentes. Posteriormente, esses agentes são agrupados de acordo com suas características, seus recursos e funcionalidades. A implementação desta arquitetura leva em consideração também algumas características de sistemas multiagente (MAS).

A arquitetura EPSCore é uma estrutura simples que auxilia no aprendizado de inteligência artificial distribuída através do uso do conceito de agentes inteligentes aplicados à manufatura. O aprendizado pode ser verificado quando, no processo de desenvolvimento do sistema, é possível ver a criação dos agentes, a comunicação, a execução das funcionalidades e a formação de coalizão entre os agentes do sistema.

A arquitetura proposta é fundamentada basicamente em sistems multiagentes e EPS, de forma que, uma característica primordial é que o foco da inteligência do processo produtivo está nos agentes que representam os produtos sendo fabricados. Isto é, o conhecimento de como se fabrica um determinado produto, é retirado dos módulos(módulos mecatrônicos) que realizam a atividade de montagem. A confecção desse produto acontece depois de interações entre os agentes do sistema.

Os agentes desenvolvidos são criados dento da plataforma, têm funcionalidades autônomas, podem se comunicar com os demais agentes do sistema, modelam e implementam comportamentos individuais e/ou sociais. Usando-se uma das formas de comportamento social, a cooperação, forma-se uma rede de agentes que têm a capacidade de resolver coletivamente os problemas de fabricação, transformando-os grandes problemas

em menores. Além disso, o sistema tem características de modularização, reconfiguração e *plug-and-produce* (é o *plug-and-play aplicado à manufatura*).

Os agentes do sistema têm somente a inteligência de seu próprio funcionamento e são implementados como agentes inteligentes que possuem a capacidade de ação no mundo físico, através de seus sensores e atuadores, capacidade de computação própria e comunicação com as outras entidades do sistema. Dessa forma, cada agente é associado a um módulo mecatrônico e, consequentemente, é chamado de agente mecatrônico. Como tem capacidade de ação no mundo físico, é chamado também de agente motor.

Os agentes inteligentes podem ainda estar associados a sistemas mecatrônicos que não têm acesso ao mundo físico, porque realizam as atividades de montagem, como no caso de agentes produto. Estes agentes, como estão associados a módulos mecatrônicos também são chamados de agentes mecatrônicos; contudo, por não precisarem acessar o mundo físico são chamados também de agentes cognitivos.

Agentes produto são um tipo específico de agente cognitivo que contém a inteligência do processo produtivo. Contudo, tal agente também não tem toda a informação necessária para a fabricação do produto, apenas a sequência de operações em termos de chamadas de *skills* (funcionalidades ou serviços) de outros agentes mecatrônicos, sejam estes outros agentes motores ou cognitivos.

Assim, nenhum dos agentes do sistema contém *toda* a informação necessária para se fabricar um determinado produto, todavia os produtos são montados em um ambiente dinâmico. A inteligência de como será montado um produto está no sistema, porém distribuída em todas as partes que o formam. Dessa forma, a integração das partes, ou melhor, a coalizão entre os agentes, é o que torna possível as atividades de montagem.

A arquitetura EPSCore é apresentada na Figura 11 e é formada por vários agentes mecatrônicos e não-mecatrônicos que são separados em quatro camadas distintas, de forma que os agentes agrupados em determinada camada tenham características comuns. As funcionalidades oferecidas são, por exemplo: registro, negociação e alocação de recursos, funcionalidades corriqueiras e úteis na interação dos agentes. Assim, podemos apresentar os tipos de agentes e suas respectivas camadas:

Camadas:

- Camada física: é formada por agentes mecatrônicos e tem a responsabilidade de perceber e de agir sobre o mundo físico através do acionamento de atuadores e sensores presentes no sistema;
- Camada cognitiva: é formada por agentes mecatrônicos que contêm inteligência capaz de coordenar outros agentes mecatrônicos através da troca de mensagens e chamadas de *skills* que acontecem no sistema;

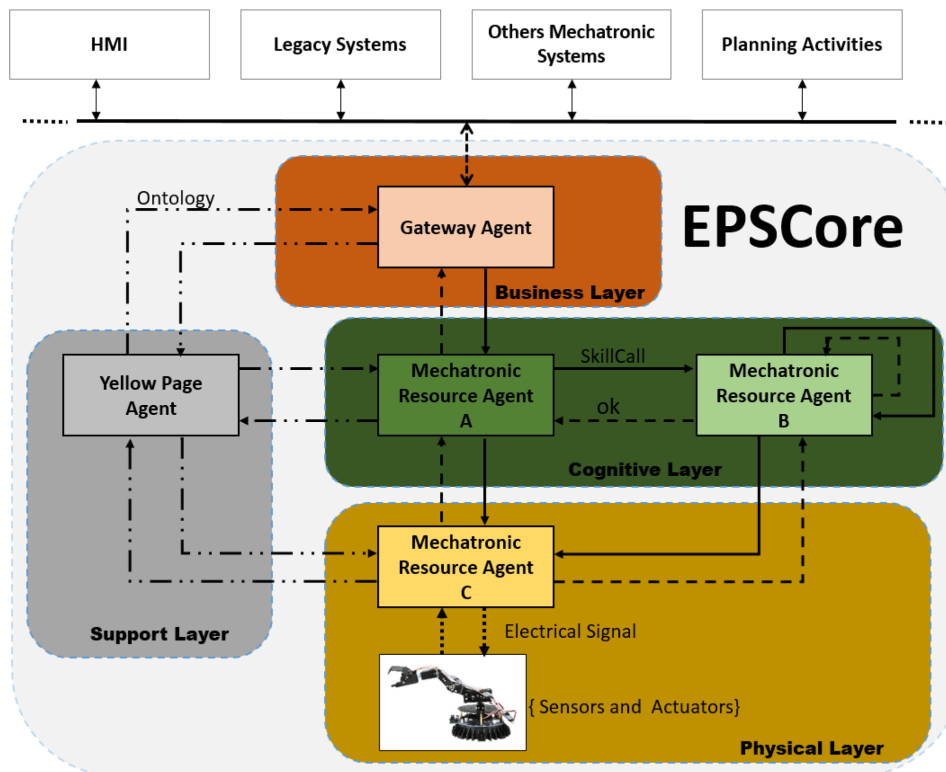


Figura 11 – Camadas da Arquitetura EPSCore

- Camada de suporte: é formada por agentes não-mecatrônicos que dão suporte a execução de serviços e descoberta de funcionalidades no sistema para os demais agentes.
- Camada de negócios: contém agentes não mecatrônicos que representam a ponte de comunicação entre a arquitetura EPSCore e os outros sistemas que podem ser desde um monitor a um outro sistema mecatrônico ou não;

Agentes:

- Agentes Mecatrônicos (*Mechatronic Agent* – MRA): agente central da plataforma, externa suas habilidades através do `YPA` e pode executar chamadas locais ou remotas de *skills*, isto é, pode servir tanto como um líder de coalizão quanto um agente de recurso. Esses agentes podem ser de dois subtipos:
  - Agente Cognitivo (*Cognitive Agent* - CA): contém uma inteligência capaz de coordenar outros agentes mecatrônicos, ou seja, é responsável pela lógica da aplicação, seleção e integração dos agentes motores ou outros agentes cognitivos. É equivalente ao agente CLA da implementação europeia, isto é, é capaz de executar *skills* remotos. Agentes cognitivos encapsulam a complexidade do

processo produtivo, deixando a granularidade mais grossa para outros agentes cognitivos.

  - Agente Motor (*Motor Agent* – MA): contém a inteligência responsável por perceber e atuar sobre o mundo físico. É equivalente ao `MRA` da implementação europeia, isto é, é capaz de executar *skills* localmente. Um agente motor especial é o agente de transporte (*Transport Agent* – TA).

  No desenho da arquitetura nós temos agentes mecatrônicos A, B e C (ver Figura 11). Os agentes A e B são cognitivos que podem realizar tanto chamada de execução local, quanto remota. O agente C é um agente do motor. Os agentes podem fornecer serviços a outros agentes do sistema na forma de *skills*.

- Agentes Não-Mecatrônicos: estão contidos na camada suporte e na camada negócios. Na proposta são apresentados dois agentes: o agente Páginas Amarelas (`YPA`) e o agente ***Gateway***(`GA`):

  - Agente de Páginas Amarelas (***Yellow Page Agent*** - `YPA`): é o agente responsável pela definição de área do sistema mecatrônico EPS. Esse agente não é mecatrônico, mas faz parte da plataforma, com uma função crucial: registro e busca de *skills*. É o serviço de descoberta de *skills* que possibilita a auto-organização de um EPS. É fundamentalmente um agente de suporte;
  - Agente *Gateway* (***Gateway Agent*** - `GA`): é o agente que une os sistemas multiagentes tradicionais com o *framework* EPS. Ele, de um lado, pode emitir, instanciar agentes mecatrônicos e chamar *skills*; por outro lado, pode comunicar-se com outros sistemas do processo produtivo que não são EPS.

A divisão em camadas existe como abordagem simplificada para facilitar a solução de problemas mais complexos partindo do princípio "dividir para conquistar". Entretanto, uma divisão grande ou em larga escala aumenta a complexidade do algoritmo e do sistema devido à necessidade de as camadas trocarem informações entre si. Dessa forma, podemos dizer, que caso ocorresse o acréscimo de agentes e de camadas, haveria um custo de tempo maior na programação e aumento da requisito computacional em virtude da necessidade de mais tipos de interações, podendo também ocorrer a inviabilização do funcionamento do sistema.

O serviço disponibilizado pelo agente é uma habilidade específica que pode ser executada tanto por *hardware* como por *software*. Há duas formas de um agente começar a execução deste serviço: a primeira, quando acontece através de uma chamada de outro agente; e a segunda, por meio de um método que executa automaticamente, conforme configuração do agente, que em geral ocorre na inicialização.

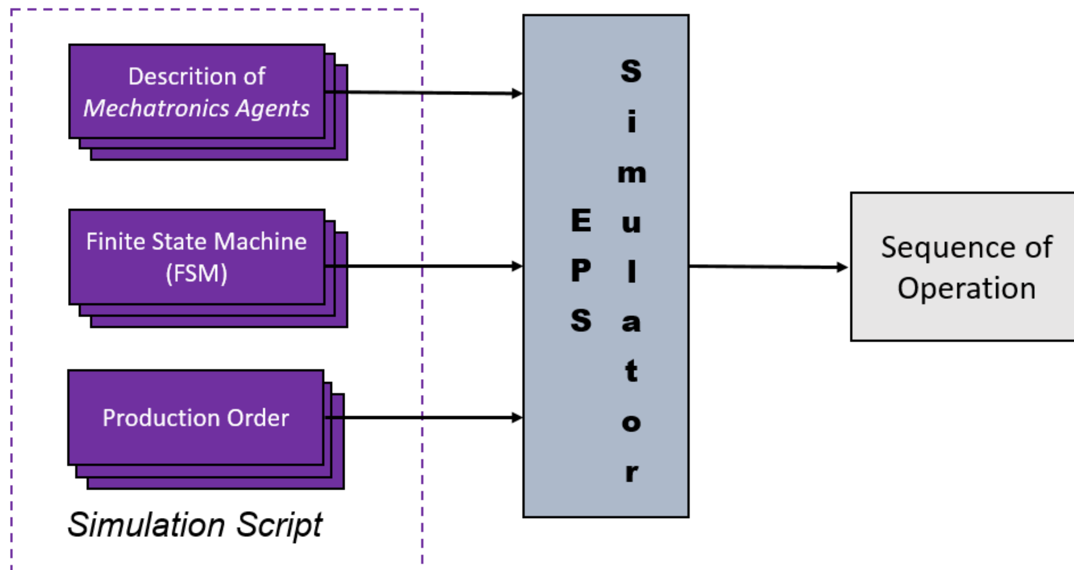


Figura 12 – Proposta do Simulador EPS.

Para fins de validação da arquitetura EPSCore, foi utilizada um protótipo de um sistema mecatrônico real. Esse protótipo é a nossa aplicação real e foi desenvolvida em conjunto um aluno de mestrado da UFAM. Através da aplicação foi avaliado o desempenho da arquitetura, comunicação e interação dos agentes. Conjuntamente com a aplicação, é proposto um simulador EPS.

4.2 Proposta do Simulador

A proposta do simulador entra com o objetivo de reproduzir o comportamento temporal dos agentes mecatrônicos de um sistema operando sob determinadas condições. Essa reprodução permite uma maior aproximação com a realidade, possibilitando a prevenção de comportamentos não desejados nas máquinas, entre outras funcionalidades. A simulação permite que se faça uma análise do sistema sem a necessidade de interferir no mesmo. Todas as mudanças e consequências, por mais profundas que sejam, ocorrerão apenas no modelo computacional, possibilitando melhorar o planejamento e a eficiência do sistema ao máximo.

Na Figura 12, é apresentada a proposta do Simulador EPS, que funcionará como um interpretador de *scripts* de simulação e terá a possibilidade de prever o comportamento do sistema, dada uma determinada condição, a fim de antecipar os efeitos produzidos por alterações ou mesmo pela evolução ou surgimento de uma situação não desejada. Em seguida serão apresentadas as entradas, que formam o *script* de simulação, e a saída do Simulador:

- Production Order: descreve uma sequência de tarefas para execução (a simulação do funcionamento do sistema), que pode ser planejada pelo usuário ou gerada automaticamente. No segundo caso, será criado um *script* com vários produtos pré-definidos ou um *script* que tem a capacidade de gerar todas as combinações de anagramas dadas as letras disponíveis no sistema;

- Description of Mechatronics Agents: especificação das características e das funcionalidades dos agentes. Por exemplo, agentes braços robóticos, esteiras, acionadores, entre outros;

- Finite State Machine (`FSM`): é uma máquina de estados finita que descreve o comportamento de um módulo mecatrônico e contém as informações relacionadas ao seu funcionamento físico, por exemplo, a máquina de estados de um braço robótico compreende os estados ligar, mover, pegar no ponto A, mover, deixar no ponto B, e finalizar. As FSMs podem ser encaradas como a descrição dos estados dos agentes mecatrônicos do sistema que, nesse caso, dado as condições pré-definidas do comportamento e tempos de execução dos módulos, o funcionamento do sistema é reproduzido para produção de uma determinada demanda.

- Sequence of operation: é o resultado do uso do Simulador EPS, em que será possível visualizar os agentes que foram criados, a execução do sequência de operação do sistema para fabricação de um produto e, futuramente, as trocas de mensagens entre os agentes. Assim, o desenvolvedor tem como retorno do uso do Simulador EPS, um *script* com a sequência de execução das operações do sistema, das chamadas de *skill* dos agentes e o funcionamento para a fabricação dos produtos planejados, tais como as informações de tempo de execução.

A máquina de estados genérica do sistema é apresentada na Figura 13 e segue as seguintes etapas:

- Quando a FSM é iniciada, ela fica no estado *"Waiting Order"* aguardando a chegada da ordem de produção, sendo que a ordem contém a definição dos produtos que serão fabricados;

- No estado *"Produces Product"* é iniciada a fabricação que pode seguir para o estado: *"Production Failure"*, caso haja algum erro no processo e *"Production Success"*, se finalizado com exito;

- Depois desses estados a ordem é finalizada e volta para o estado *"Waiting Order"*;

- A máquina de estados é finalizada quando o sistema é desligado.

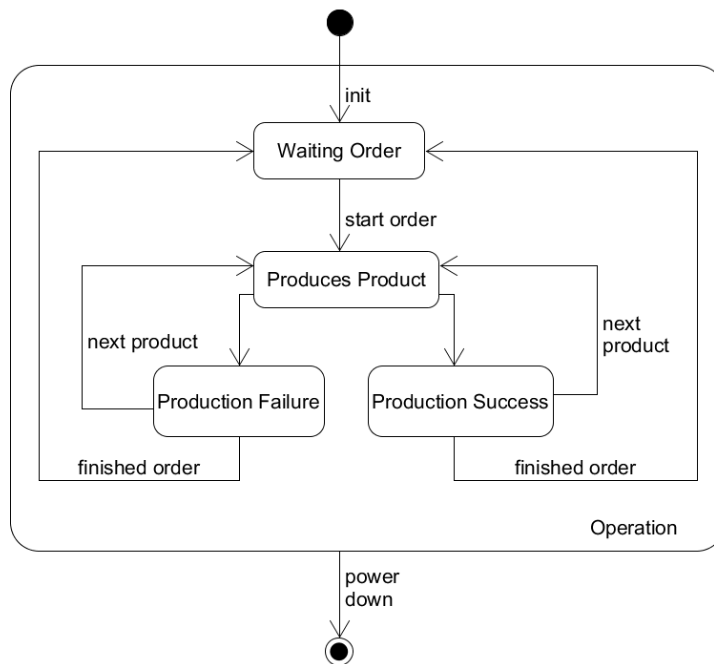


Figura 13 – Máquina de Estados Genérica

O objetivo do Simulador EPS é desenvolver um sistema mecatrônico e testá-lo em duas situações que podem acontecer: uma antes de implementar o sistema real e a outra, quando o sistema existe e é desejável implementar melhorias ou estudar o comportamento dos agentes do sistema. Existe ainda a possibilidade de posteriormente, reusar o algoritmo utilizado na simulação para implementação dos agentes. Assim, podemos dizer, que o uso do Simulador proporcionará redução de custos pela redução temporal no funcionamento dos agentes e melhoria de desempenho do sistema.

Na Figura 14 é mostrado como funciona o simulador, partindo da premissa de que o sistema inicia a partir da leitura do XML que contém as informações que direcionarão o Simulador EPS para reproduzir o funcionamento do sistema. As seguintes etapas são seguidas:

- Etapa 01 - O XML é decodificado pelo agente `Simulator` e é o responsável pela execução do *script* de simulação;
- Etapa 02 - O agente Simulador instancia os MRAs motores e suas máquinas de estados. Depois, instancia os agentes MRAs cognitivos. Em seguida, instancia os agentes produto, que são MRAs onde é definido o processo produtivo a ser realizado, em termos de chamadas de *skill* suportadas pela plataforma EPSCore.

  A cada novo agente produto instanciado, este começa a executar o seu processo produtivo e o Simulador apanha as saídas durante a execução, isto é, sua sequência de operação;

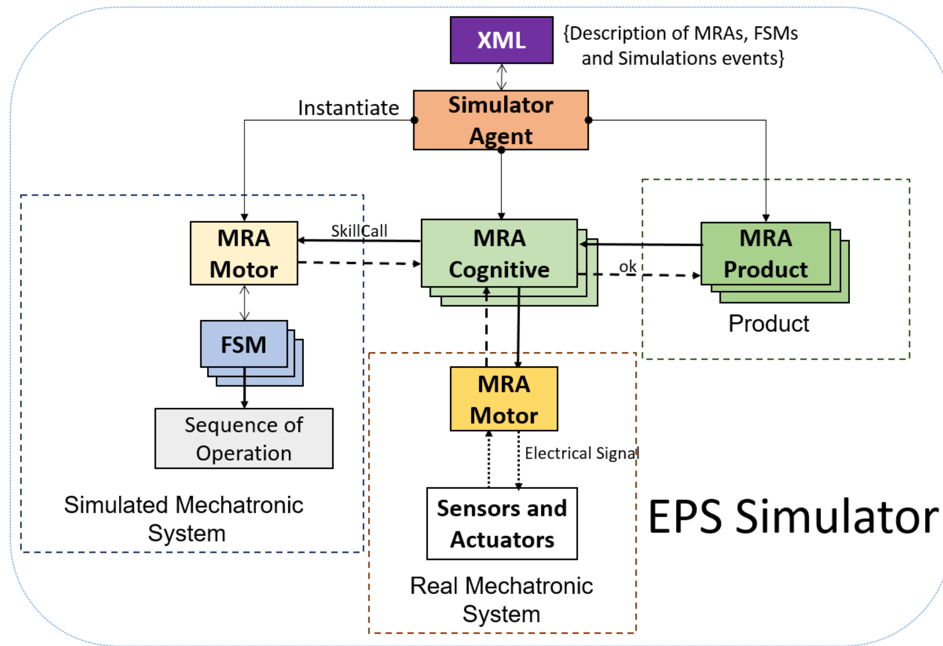


Figura 14 – Funcionamento do Simulador EPS.

- Etapa 03 - De acordo com a ordem de produção, é iniciado uma sequência para a fabricação dos produtos. A forma de comunicação entre os produtos e os MRAs é realizado por meio de chamadas de *skill*. Ao se descrever uma ordem de produção, são definidas as sequências de operação; há duas formas de operação possíveis que o programador pode definir:

  – *caminho do simulador*: definido na Figura 14 pelo retângulo tracejado azul, acontece quando os MRAs do sistema fazem chamadas de *skills* no agente motor simulado.

    Este agente é responsável por realizar as ações de fabricação dos produtos. Este age em máquinas de estado que descrevem o comportamento temporal dos agentes mecatrônicos de um determinando sistema, ou seja, a máquina de estados simula o ciclo de funcionamento, em termos de seu tempo de execução, de um hardware real;

  – *caminho do sistema real*: definido na Figura 14 pelo retângulo tracejado laranja. É quando os MRAs do sistema fazem chamada de *skills* no agente motor do sistema real. Este agente se comunica, por meio de sinais elétricos, com os sensores e atuadores de um sistema mecatrônico real e atua para que sejam fabricados os produtos.

Do ponto de vista dos demais agentes, se comunicar com o agente motor simulado ou o agente motor do sistema real não faz diferença, pois essa comunicação ocorre por

meio de uma chamada de *skill*. Assim, podemos dizer que o Simulador EPS está simulando somente os tempos de funcionamento dos agentes mecatrônicos através de sua máquina de estados; portanto, estamos simulando somente o *hardware* como se fosse uma máquina de estados finita que recebe chamada de *skills*.

4.3 Etapas para o Desenvolvimento

No capítulo seguinte serão detalhadas as seguintes etapas:

- Etapa 01: processo de desenvolvimento da arquitetura e dos agentes, sua comunicação no sistema e funcionalidades disponibilizadas;
- Etapa 02: desenvolvimento da ontologia com o objetivo de padronizar a comunicação no sistema, como é usada as *skills* e a descrição das classes da EPSCore e de seus agentes;
- Etapa 03: descrição do agente de integração responsável pela ponte entre outros sistemas e a arquitetura ESCore;
- Etapa 04: aplicação da arquitetura EPSCore usada para validar uma das propostas, apresentação do escopo e do funcionamento dessa aplicação;
- Etapa 05: uso do simulador EPS com sua descrição de funcionamento e mostrar o objetivo, que é a reprodução do comportamento temporal dos agentes mecatrônicos do sistema que foi simulado.

5 Implementação

O conhecimento e a tecnologia devem ser disseminados no campo mais amplamente possível, a fim de permitir que os desenvolvedores apresentem propostas que possam gerar ferramentais úteis à manufatura. Assim, com o objetivo de apresentar uma mudança paradigmática na manufatura e esta passe a utilizar os conceitos trazidos com os EPS é necessário que sejam desenvolvidas ferramentas. Neste capítulo, será descrita a arquitetura EPSCore, sua implementação denominada SIAPE e o simulador EPS.

5.1 *Skill* na arquitetura EPSCore

Na figura 15, é apresentado um diagrama de classes parcial contendo a definição de *skill* para arquitetura EPSCore. `SkillBase`, `Skill` e `SkillTemplate` são as classes que representam repectivamente uma classe abstrata para representação de *sklls*, uma classe abstrata para a construção de *skills* e uma classe concreta que representa um *skill* durante seu registro, busca e execução.

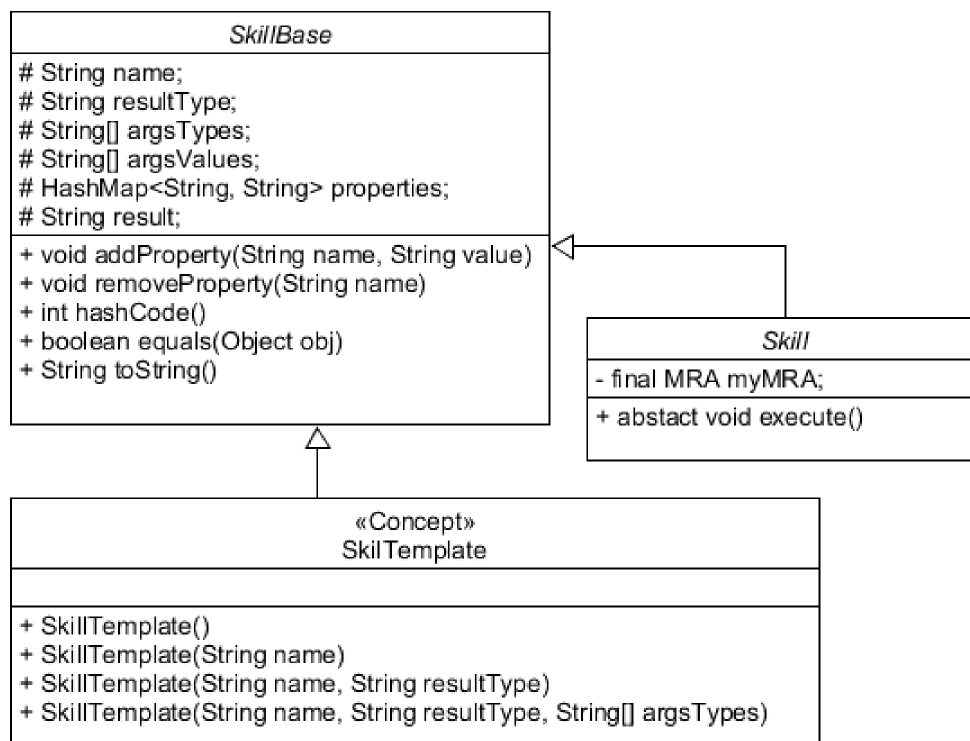


Figura 15 – *Skills* definidos na arquitetura EPSCore.

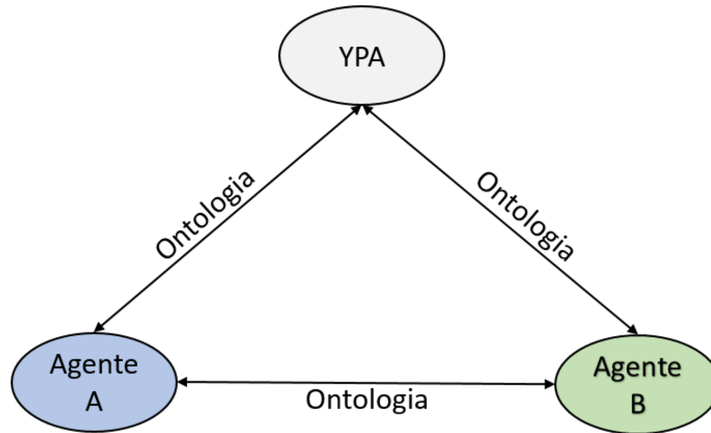


Figura 16 – Ontologia para EPS.

- a classe `SkillBase` é definida como a classe principal que contém um padrão de informações referentes ao nome, ao tipo, aos argumentos, aos tipos de retorno e às propriedades das habilidades aqui padronizadas;
- a subclasse `Skill` é usada para realizar a chamada de serviços dos agentes do sistema. Através dela é possível executar as *skills* dos agentes do sistema;
- a subclasse `SkillTemplate` é um conceito que define um conjunto de *templates* (padrões) para realizar a troca de informações de determinadas *skills* no sistema.

5.2 Ontologia

Com base na referencia teórica, apresentada neste trabalho, foi desenvolvida uma ontologia para EPS em que foi definida uma estrutura comum de significados para serem usados pelos agentes na comunicação sem que aconteça a possibilidade de ambiguidades. Assim, foi criado um vocabulário e semântica próprios para a troca de mensagens na arquitetura EPSCore. Na Figura 16, é apresentada como funciona basicamente a comunicação dos agentes:

Na implementação, são criadas classes para usar os conceitos e ações dos agentes. Neste caso, um agente que foi criado na arquitetura pode conter ou solicitar uma ou mais ações que serão executadas quando as condições do agente estão satisfeitas e o agente é acionado (condições estas que podem ser mensagem padrão ou agente disponível). Um agente pode conter qualquer combinação de ações.

Os conceitos (*Concepts*) e as ações (*AgentActions*) da arquitetura são baseados na classe `BeanOntology`, que é uma extensão da classe `Ontology`, que permite criar automaticamente todos os esquemas de uma ontologia Jade a partir das classes Java que

representam os elementos ontológicos (classes ontológicas). Os conceitos e as ações da ontologia usados no EPSCore estão enumerados abaixo:

- Ontologia - define a linguagem padrão usada na arquitetura que é a ***EPSOntolgy***;
- *Concepts* (Conceito) - é um `MRAinfo` que representa um conjunto de informações sobre determinado agente e o `SkillTemplate` que representa um agrupamento padrão de informações acerca das *skills* de um agente;
- *AgentAction* - são ações possíveis definidas para realização de determinado serviço que nesse caso podem ser: `Register` - é encarregado de registrar os agentes no agente de páginas amarelas; `Unregister` - faz o cancelamento do registro dos agentes no agente de páginas amarelas; `Search` - é encarregado da pesquisa no agente de páginas amarelas a fim de descobrir o *Agent Idenfication* (`AID`) de um agente que executa determinada funcionalidade ou simplesmente consultar os serviço disponíveis nas páginas amarelas; `Execute` - é a ação responsável pela execução da *skill* do agente.

Para usar a ontologia EPS é necessário criar as classes Java (`Beans`) que descrevem os conceitos, ações de agente e predicados relevantes para o domínio abordado. Depois, antes de um agente, pode realmente usar a ontologia definida e o idioma selecionado. É preciso que a ontologia seja registrada para o gerenciador de conteúdo do agente. Esta operação é tipicamente (mas não necessariamente) realizada durante a instalação do agente.

Em uma conversa entre os dois agentes a ontologia segue um protocolo muito simples. Por exemplo: para criar uma conta ou fazer uma operação, o agente cliente envia uma mensagem de solicitação para o agente servidor. O agente servidor responde com um `"INFORM"` depois de processar o pedido ou com um `"NOT-UNDERSTOOD"` se não puder decodificar o conteúdo da mensagem. Para consultar informações sobre uma conta específica, o agente cliente envia uma `"QUERY-REF"` para o agente servidor que responde com um `"INFORM"` após o processamento da consulta ou com um `"NOT-UNDERSTOOD"` se não puder decodificar o conteúdo da mensagem. Essa comunicação é padronizada pela ontologia, de forma que todos os agentes falem a mesma língua.

Na Figura 17 é apresentado um diagrama que contém classes e *AgentActions*:

- `EPSOntology` que é um `BeamOntology` responsável pelo definição, registro e instanciação da ontologia;
- `Unregister` é um *AgentAction* encarregado de cancelar o registro dos agentes mecatrônicos dentro do agente páginas amarelas;

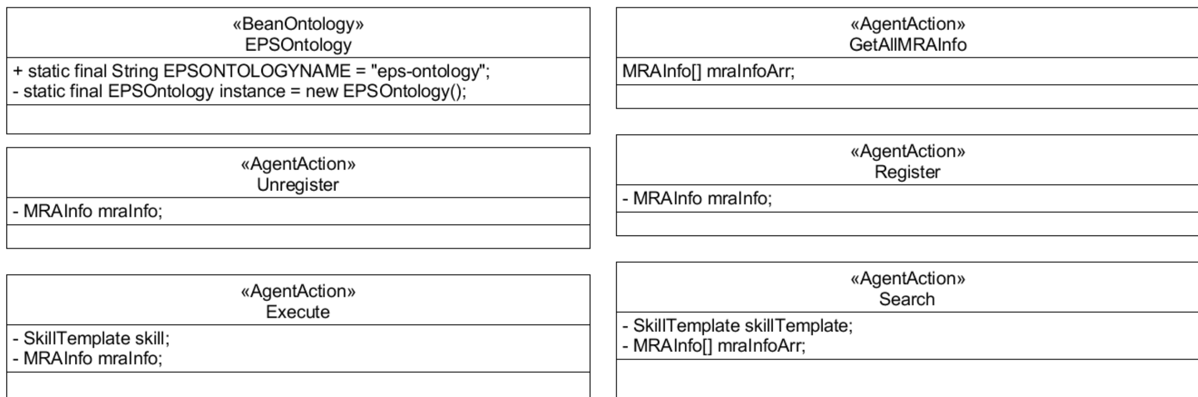


Figura 17 – Classe UML - Ontologia para EPS - ***BeanOntology*** e *AgentActions*.

- `Execute` é um *AgentAction* responsável pela definição dos `SkillTemplate` e posterior chamada e execução dos *skills*;

- `Register` responsável pelo registro dos agentes mecatrônicos no agente suporte páginas amarelas através de informações contidas no Mrainfo do mesmo; `GetAllMRAInfo` que agrupa as informações contidas no mrainfo de todos os agentes instanciados;

- `Search` que, de acordo com os argumentos e `SkillTemplate`, disponibiliza uma busca às *skills* (funcionalidades) dos agentes registrados no sistema. Assim, percebe-se que o conceito de *skills* permeia, então, todo o arcabouço da execução flexível de um sistema de manufatura baseado em EPS.

5.3 Arquitetura EPSCore

No desenvolvimento da arquitetura EPSCore, uma vez que foi inspirada na IADE, foi usado o `JADE` que é uma implementação de `FIPA` e um *framework* para desenvolvimento de agentes. O `JADE` já disponibiliza o suporte da comunicação provida pelos protocolos `FIPA`, fornece a arquitetura básica para o desenvolvimento de agentes, gerencia a plataforma, os contêineres e oferece vários outros serviços e funcionalidades como a publicação e subscrição de serviços.

O funcionamento simplificado, otimizado também do sistema, deve-se ao fato de que na divisão por camadas, cada camada é responsável por uma única função (ou poucas funções) separadas de acordo com os serviços disponibilizados, de forma que, as atividades que requeiram maior deliberação vão para o agente cognitivo. As atividades de acesso ao *hardware* (nesse caso o monitoramento de sensores e o uso dos atuadores) vão para o agente motor, proporcionando uma boa performance, ganho com a flexibilidade, entre outras vantagens.

A camada física contém agentes mecatrônicos que realizam o equivalente ao agente de recurso da arquitetura IADE, isto é, eles são capazes de executar serviços localmente ou remotamente. Um agente mecatrônico pode fornecer serviços a outros agentes do sistema na forma de *skills*. Nesta camada os agentes mecatrônicos são denominados de agentes motores (*Motor Agent* – MA).

A camada cognitiva ou camada de processo contém agentes que realizam o equivalente ao CLA da IADE, isto é, são capazes de realizar serviços locais ou remotos. Nessa camada, o agente mecatrônico fornece serviços a outros agentes do sistema na forma de *skills*, executados através de chamadas de outros *skills* locais ou remotos. Tais agentes são denominados agentes cognitivos (*Cognitive Agent* - CA). Um subtipo de agente cognitivo é o agente responsável pela montagem dos produtos, chamado de *Product Agent*.

A camada de negócios contém agentes que interagem com outros sistemas, por exemplo, com uma `IHM`, sistemas legados (*Legacy Systems*), ou outro sistema de mecatrônico, com atividades de planeamento, e assim sucessivamente. Um agente dessa camada é o `(`***`Gateway Agent`***` - GA)` responsável por unir os sistemas multiagente tradicionais com `EPS`, interpretar as informações do operador e instanciar os agentes mecatrônicos na arquitetura EPSCore.

Na camada de suporte, existe um agente de páginas amarelas (`YPA`) que representa a área do processo, toda vez que o sistema é iniciado o `YPA` iniciará junto. Ele é um agente não mecatrônico responsável pela funcionalidades de publicação de serviços dos agentes. Todos os agentes, logo que nascem, registram-se no `YPA`.

A arquitetura EPSCore é um EPS que apresenta auto-organização, isto é, o sistema é capaz de reconhecer os agentes que estiverem ativos em determinado momento e formar a sociedade de agentes necessária para a execução de algum objetivo no sistema. EPSCore, então, possui uma característica chamada de *plug-and-produce* (plugar e produzir), em que cada módulo do sistema pode ser retirado do, ou colocado no, sistema (*on-line*) como parte integrante de seu funcionamento normal. E suas classes são apresentadas através do diagrama de classes na Figura 18.

As classes que compõem a arquitetura estão divididas em dois grandes blocos: um que contém as classes do `core` e o outro que a ontologia `eps.ontology` e classes auxiliares. As classes são:

- `MRA`: define um agente mecatrônico, ou seja, para se criar um agente mecatrônico é necessário que a classe em questão, seja filha de `MRA`;
- `YPA`: é a classe que define o agente de páginas amarelas. É definido como área para toda a arquitetura EPSCore e oferece serviços como: registro, pesquisa e cancelamento de registro;

- `EPSOntology`: define a ontologia para a arquitetura EPSCore. A ontologia é registrada como "`eps.ontology`";
- `Skill`: define um *skill*. Ela é uma abstração do método de execução remota. Nele é implementado um método que é a ação do *skill*. Esta ação é executada logo que a *skill* é requerida;
- `SkillBase`: é uma classe para *skills*. Ela representa o modelo de *skills* usado na arquitetura;
- `SkillTemplate`: é um padrão/template para o registro e a pesquisa de *skills* na arquitetura. Estes serviços são solicitados/realizados através do `YPA`;
- `MRAInfo`: são informações específicas sobre os agentes mecatrônicos. Contém as seguintes informações:
  - `Name`: uma identificação única do agente ou módulo mecatrônico;
  - `SkillTemplate`: uma lista de *SkillTemplates* do agente ou módulo mecatrônico;
  - `Poperties`: uma lista de propriedades que podem ser mecânicas, elétricas ou atributos lógicos do agente ou módulo mecatrônico.
- `MRAServices`: é chamado esse método para criar um agente de execução remota para o `MRA`. Essa classe é composta por:
  - `Agent`: o agente que requer a execução do *skill*;
  - `MRAInfo`: `MRA` que executa o *skill*;
  - `St`: é o *template* do *skill* que será executado no agente alvo;
  - `Return`: o retorno da *skill* executada.
- `YPAServices`: é chamado esse método para criar um agente de execução remota para o `YPA`. Essa classe é composta por:
  - ***`Register`***: faz o registro de todos os agentes que são criados;
  - ***`Search`***: serviço disponibilizado a outros agentes do sistemas para consulta das funcionalidades ou serviços disponibilizados por outros agentes do sistema;
  - ***`Unregister`***: é responsável pelo cancelamento de registro dos agentes na arquitetura.
- `Product`: é um produto genérico. Implementa o método ***`produce()`*** que ativa um plano de produção;
- `Util`: é a classe responsável pela mudança/conversão da *skill*. Ela realiza a mudança de ***`skillTemplate`*** para *skill*.

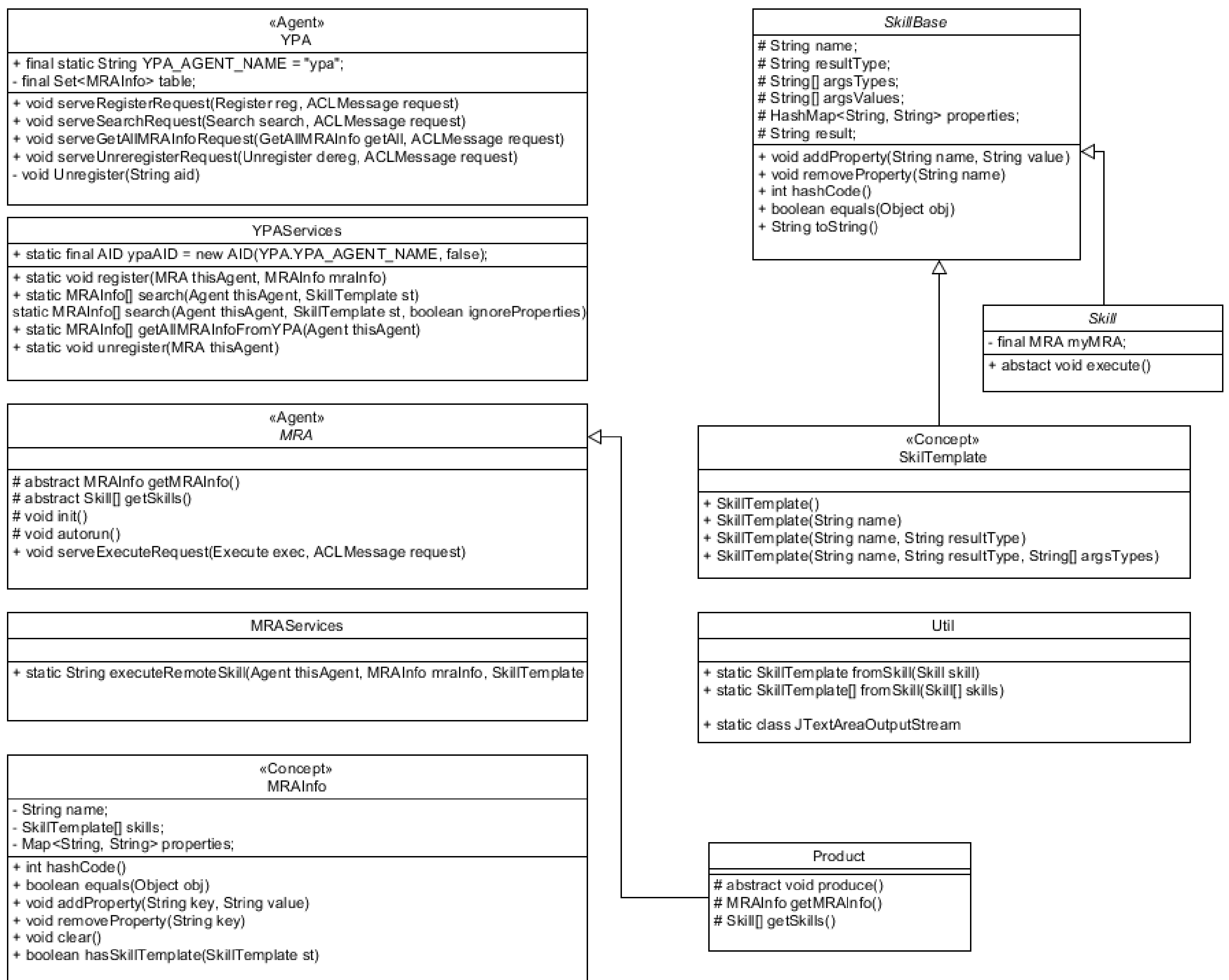


Figura 18 – Classes da Arquitetura EPSCore

5.3.1 Definição dos Agentes

`YPA` - *Yellow page agent*

É um agente não mecatrônico que externa serviços especializados de páginas amarelas. O YPA armazena os `AIDs` e serviços dos demais agentes da plataforma que, assim que criados, devem se registrar no agente de páginas amarelas (YPA).

Com o registro o YPA tem as informações de `AID` e serviço que este agente pode realizar. Dentre as funcionalidades deste agente destaca-se a consulta, a qual é solicitada por um agente do sistema com a finalidade de obter informação relativa a quantos agentes realizam um determinado serviço ou qual ou quais agentes executam esse mesmo serviço.

Os serviços disponíveis no YPA são: registro (`Register`), cancelamento de registro (`UnRegister`) e procura/pesquisa (`Search`). Com o propósito de facilitar o desenvolvimento

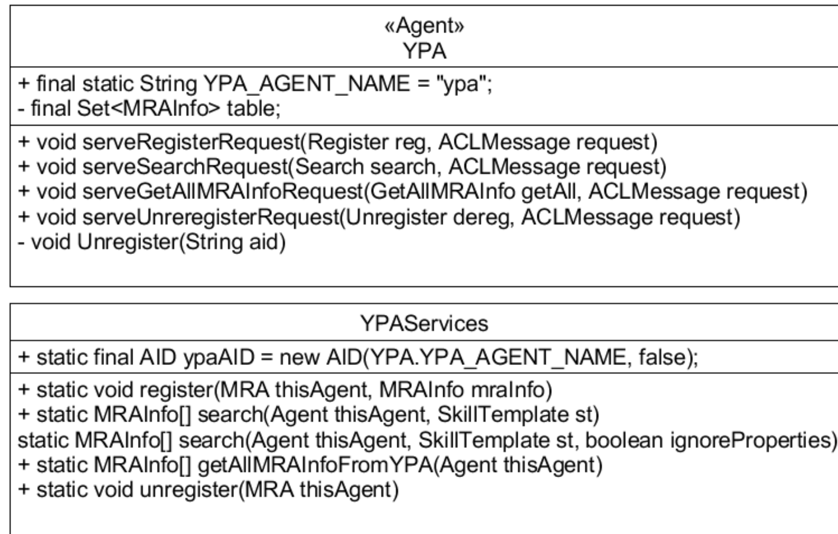


Figura 19 – Diagrama de Classes em UML do YPA.

foi criada uma classe extra, vinculada ao YPA, denominada `YPAServices`. Sendo que esta classe disponibiliza chamadas dos serviços e repassa informações ao YPA para este cumprir uma determinada requisição de serviço.

A Figura 19 apresenta o diagrama de classes do YPA e seu auxiliar o YPAServices: quando um agente utiliza um dos serviços que o YPA disponibiliza, a comunicação entre dois agentes acontece através de mensagens FIPA. É usado `ACLmessage` para realizar requisições (`request`), aceites (`agree`) ou recusas (`refuse`) e informes(`inform`).

Por exemplo, um agente “A” será definido cliente e o outro agente “B” o agente que realiza o serviço de que “A” precisa. Portanto, o agente “A” manda uma mensagem do tipo ***request*** para solicitação de pesquisa ao YPA com conteúdo “ quem executa o serviço de transporte?”. O `YPA` manda um ***inform*** com o `AID` de um ou mais agentes que executam o serviço de transporte. Cabe ao agente “A” entrar em contato com o agente “B” que executa o serviço desejado e fazer a solicitação de execução desta funcionalidade.

Um `request` é enviado pelo agente “A” para o agente “B” para que este execute o serviço de transporte. Cabe ao agente “B” responder um `agree` ou `refuse` e, posteriormente, um `inform` é mandado se for aceito e um `failure` se a resposta for negada.

Este agente está presente dentro da camada suporte na arquitetura EPSCore e representa a definição de área do sistema, ou seja, a abrangência, o YPA delimita o ambiente em que os agentes estão interagindo entre si. Todos os agentes ao serem criados são automaticamente registrados no YPA, assim como, quando morrem, têm seus registros cancelados.

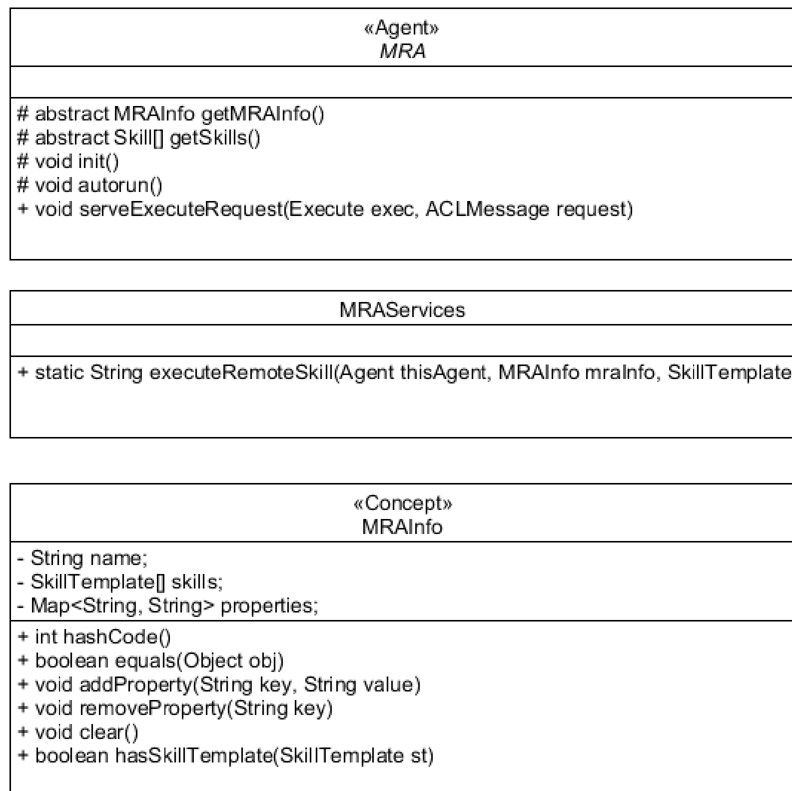


Figura 20 – Diagrama de classes em UML do MRA.

MRA - *Mechatronic Resource Agent*

É uma classe que define o agente mecatrônico que expõe e executa as *skills*. É um agente abstrato, pois ele não tem a definição de métodos como `GetMRAinfo` e `GetSkills`, sendo que estes, são chamados e definidos pelos filhos de MRA.

O MRA se estende da classe `Agent` e implementa chamadas de serviços como: registro no YPA e registro de ontologia. É acompanhado de uma classe auxiliar chamada `MRAServices` que tem um método que é responsável pela chamada de execução remota de *skills*.

As classes MRA e MRAServices fazem a ponte entre *hardware* e *software*, ou seja, conexão entre a parte física representada pelos atuadores e a parte lógica que é o algoritmo. Assim, ao ser solicitada a execução de uma determinada *skill* essas farão um link entre o agente mecatrônico e a programação, ou mais precisamente, apresenta a definição de chamada e execução de *skills* de forma local ou remota. Na Figura 20, é apresentado o diagrama de classes do `MRA`:

De acordo com o desenvolvimento da arquitetura, o MRA pode ser de dois tipos: cognitivo e motor. O cognitivo é um agente mecatrônico que auxilia a execução do agente motor através, por exemplo, de um método que contém um plano de execução/processo

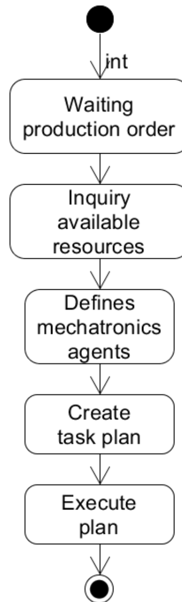


Figura 21 – Máquina de estados do agente produto.

para realizar uma sequencia de serviços. O motor é também um agente mecatrônico que realiza a execução do serviço ou da *skill* de acordo com o plano definido pelo agente cognitivo.

Um agente cognitivo especial é o agente Produto (Product Agent – PA), o qual contém a inteligência de montagem do produto (isto é, seu processo produtivo), e pode realizar, em termos de chamadas de *skills*, requisições em agentes cognitivos ou agentes motores. Na figura 21, é apresentada a máquina de estados simplificada do agente produto.

Na aplicação e simulação, experimentadas neste trabalho, a partir da arquitetura EPSCore foram criados diversos agentes filhos de MRA, tais como: `Product` e `AcHw`. O primeiro é um agente cognitivo que representa o produto; e o segundo é o agente motor através do qual a arquitetura e o sistema agirão no mundo físico, por meio de sensores e atuadores, ou seja, ele representa a conexão entre *hardware* e *software* ou o algoritmo e sistema real.

- `Product`: é uma classe que é filha de MRA, que é responsável pela definição de um agente que representará o produto. Ele é composto por dois métodos:
  - *produce*(): é acionada assim que é feita a chamada de *skill* no sistema. Ela tem execução automática e é responsável pela execução da funcionalidade de acordo com as informações recebidas com o `MRAInfo`. Esse método significa basicamente "produzir". É constituído de uma sequência que executa a funcionalidade do agente físico e no caso, se fosse na simulação, seria em uma máquina de estados;

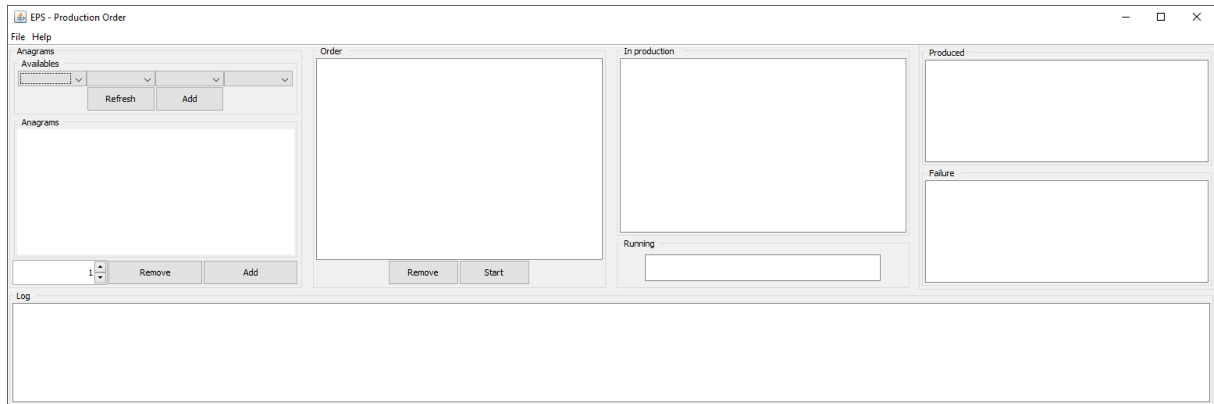


Figura 22 – Interface do `OrderAgent`.

- `GetMRAInfo`: contém informações específicas do agente que terá o método *produce*() executado, ou seja, do agente que está realizando a chamada de execução da *skill*.

- `AcHw`: é o agente que atua no mundo físico. É responsável pelo acionamento de sensores e atuadores (por exemplo: esteiras, braços robóticos, servomecanismos, entre outros).

Gateway Agent (GA)

O agente *Gateway* é o responsável pela integração do EPSCore em um sistema de agentes tradicional e/ou a outros sistemas externos. Para isso, ele contém a capacidade de realizar instanciação de agentes mecatrônicos e chamadas de *skills* nestes agentes.

Na arquitetura EPSCore, o `GA` é um ***`GuiAgent`*** e recebe o nome de ***`OrderAgent`***. Ele é um agente não mecatrônico que é responsável pela integração da arquitetura com sistemas como uma interface, sistemas mecatrônicos e/ou sistemas legados.

Na linguagem de programação Java, uma `GUI` é executada em seu próprio segmento (o evento de envio de *thread*) que lhe permite manipular e reagir rapidamente aos eventos que são gerados sempre que o usuário interage com a interface gráfica através de um componente como pressionar um botão ou redimensionar a janela.

A IHM do agente ***`OrderAgent`*** é apresentado na Figura 22. Este agente tem uma interface desenvolvida com a finalidade de facilitar a interação entre a arquitetura e a IHM. A facilidade se deve ao fato de que na interface está disponível funcionalidades como consulta de agentes do sistema, ordem de produção, produtos que em fabricação, finalizados com sucesso e com falha.

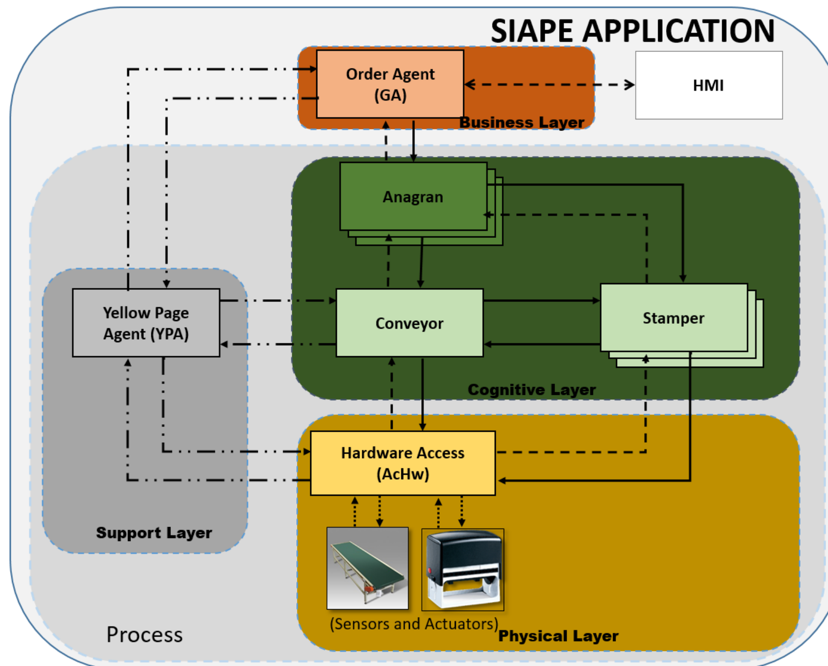


Figura 23 – Aplicação SIAPE

Ao ser instanciado o ***OrderAgent*** se registra na ontologia ***EPSOntology*** e no `YPA`. Isto é necessário para padronizar as informações (mensagens) trocadas entre eles e para estarem em uma área comum. Os seguintes métodos estão disponíveis:

- ***RefreshSkills***: consulta quais são os agentes presentes no sistema e quais são suas funcionalidades. No caso da aplicação, ele fornece os módulos que estão disponíveis para iniciar a produção;
- ***StartProduce***: da um *start* para instanciamento e produção de produtos(No caso da aplicação, anagramas).

5.4 Aplicação da Arquitetura EPSCore

Na Figura 23 é apresentado o desenho do *Sistema Inteligente Ágil de Processo Evolutivo* (SIAPE), que é uma aplicação da arquitetura EPSCore, ou seja, é um protótipo de um sistema mecatrônico que usa esta arquitetura. Esse protótipo foi construído pelo aluno do programa de pós-graduação em Engenharia Elétrica (PPGEE-UFAM), Hiram Carlos do Amaral, no trabalho (AMARAL, 2016).

O SIAPE é um EPS e, como todos os demais projetos de sistemas mecatrônicos, os agentes foram desenvolvidos com base em um protótipo real. Este protótipo visa à adaptação, à demanda e à evolução do sistema produtivo de acordo com a modificações

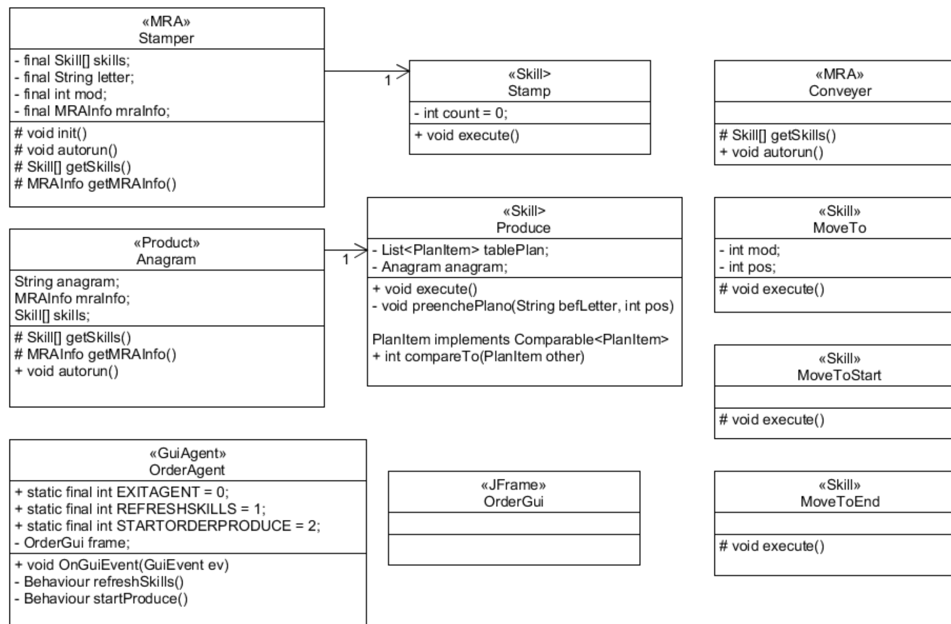


Figura 24 – Diagrama de Classe parcial da aplicação SIAPE

do produto. É composto por uma esteira, módulos carimbadores, paletes (produto) e sensores.

O objetivo do SIAPE é a produção de palavras (anagramas) que são produzidos de acordo com as letras disponíveis no sistema. Dada uma ordem de produção é iniciada a montagem das palavras ou anagramas que serão produzidos através de seus módulos carimbadores, esteiras, fontes e outros sensores e atuadores. Através da combinação das letras disponíveis no sistema é possível fabricar o produto "UFAM", bem como "UTAM", "MATU", "FT", "FAMA" etc.

Na Figura 24 é apresentado o diagrama genérico de classes em UML, que foi desenvolvido como aplicação da arquitetura EPSCore, correspondendo às classes dos agentes do sistema: `OrderAgent`, `Stamper`, `Conveyor` e `Anagran` e seus respectivos *skills*.

Além disso, o sistema foi desenvolvido para que cada módulo pudesse ser retirado ou colocado como parte integrante do seu funcionamento normal. O ***Anagran*** é limitado a 4 caracteres e devem ser inseridos na interface do sistema para que este internamente crie uma lista de tarefas, organize e depois envie uma ordem de serviço para o sistema produzir o ***Anagran***.

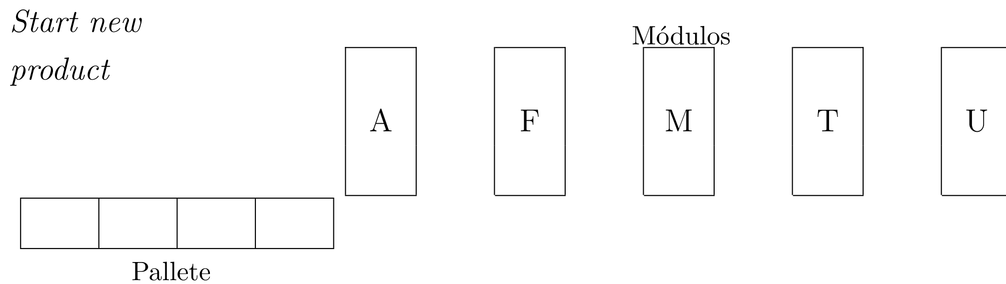
A ordem de execução do sistema pode ser descrita por:

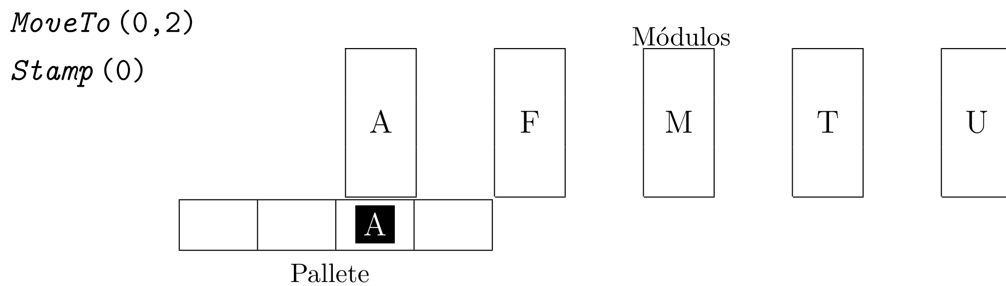
- Etapa 01: o `Anagran` verifica quais módulos estão disponíveis no sistema. Se algum módulo estiver faltando, a implementação feita considera uma falha e o produto não é montado;

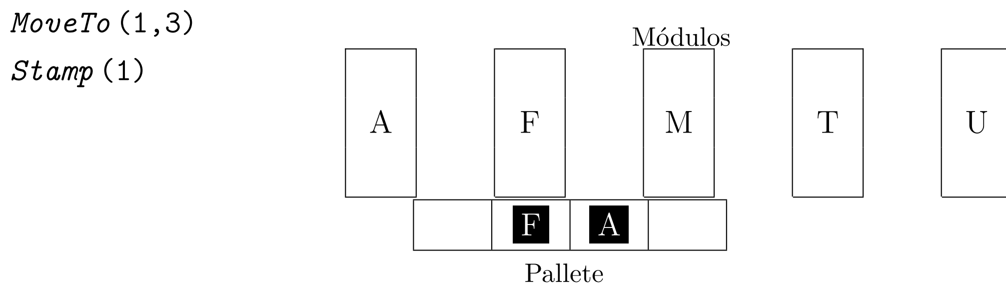
- Etapa 02: levando em consideração que a esteira realiza um descolamento seriado, o ***Anagran*** passa a verificar a ordem em que os agentes se encontram e planeja qual a ordem mais apropriada para a realização do processo produtivo (serviço este disponível na arquitetura EPSCore, no agente *product*);

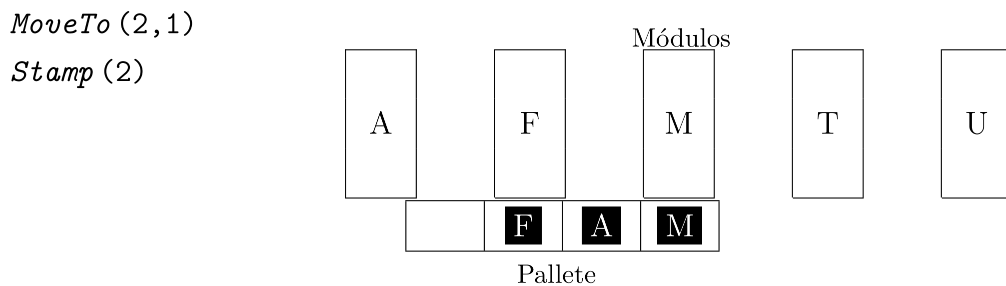
- Etapa 03: começa a execução do processo produtivo.

Na Figura abaixo, é apresentado o funcionamento da aplicação SIAPE para que seja produzido o anagrama UFAM:

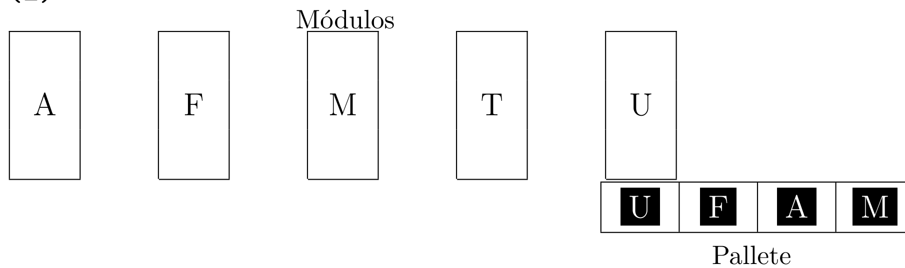








```
MoveTo(2,1)
Stamp(2)
```



5.5 Simulador EPS

A simulação computacional de sistemas, ou apenas simulação, consiste assim, na utilização de certas técnicas matemáticas, empregadas em computadores, as quais permitem imitar o funcionamento de, praticamente, qualquer tipo de operação ou processo do mundo real. Ou seja, é o estudo do comportamento de sistemas reais através do exercício de modelos. Assim, o Simulador EPS será aplicado no protótipo mecatrônico (SIAPE), construído como aplicação da arquitetura EPSCore.

O simulador EPS reproduz o funcionamento do SIAPE e cria seus respectivos agentes mecatrônicos: módulos carimbadores, esteira e outros atuadores. Ele é encarregado de demonstrar a representação do funcionamento temporal dos agentes mecatrônicos e suas respectivas comunicações e interações e, de acordo com parâmetros do projeto, proporciona que o sistema, na fase de implementação real, tenha resultados otimizados e a possibilidade de análise de funcionamento e testes de novas configurações como: inclusão de carimbos, adição de esteiras e/ou sensores.

Através do estudo de modificações e dos testes realizados no Simulador EPS, será possível visualizar, por exemplo, uma forma geral de auto-organização e a emergência no sistema e, proporcionar também o reuso da codificação, de forma que esta seja implementada no sistema mecatrônico real mantendo os resultados do comportamento simulado e somente modificando onde será aplicada a chamada de *skill*. Na simulação, a chamada de *skill* é realizada no agente Motor simulado e após todos os testes no agente Motor real.

Foram feitos testes na simulação para prever a evolução da aplicação SIAPE, como adição de mais módulos carimbadores, esteiras, sensores e outros, de forma que o sistema pudesse produzir uma maior variedade de produtos e aumentar a rede de comunicação e interação dos agentes. Se essa evolução fosse realizada no sistema físico acarretaria em gasto de tempo e demandaria muitos custos. Assim, o Simulador EPS é uma ferramenta que possibilita desde a reprodução do comportamento temporal dos agentes até uma análise detalhada do funcionamento de sistemas e possibilidade de ajuste do desempenho,

de acordo com parâmetros desejados, chegando à otimização, melhoria e evolução do sistema. Isso tudo, se planejado, antes do desenvolvimento do sistema real.

Ao se iniciar o Simulador EPS, este começa a realizar o seu processo produtivo, isto é, o conjunto de instruções que se tem para visualizar a interação dos agentes. Antes da simulação, é criado um XML do sistema real que contém desde os agentes mecatrônicos a serem simulados até a ordem de produção. Em seguida, é criado um agente simulador que executará a leitura e a decodificação do XML. Depois disso, serão definidos os requisitos do sistema e criação da máquina de estados. Por fim, depois de todas as interações, é gerado um novo arquivo com a sequência de operação do sistema. Independente da aplicação, ou seja, do sistema mecatrônico real escolhido, o simulador sempre segue essa sequência de operações.

5.5.1 `Agent EPS Simulator`

O agente simulador é responsável pela interpretação do XML que contém o *script de execução* responsável pela descrição do funcionamento do sistema e fabricação dos produto. Neste XML, estão descritos todos os agentes mecatrônicos simulados, cognitivos e motores do sistema, as *skills* desses agentes e as etapas para realização do processo produtivo .

O Simulador EPS tem como entrada diversas informações relativas ao sistema que será simulado, aos agentes deste sistema e as ordens de produção (produtos). O agente simulador obtém informações sobre os tempos de espera e execução, de funcionamento do módulo, da esteira, de resposta dos sensores através da decodificação do XML, ou seja, o tempo que leva para execução das *skills* desses agentes. Todas essas informações tornam o processo simulado o mais realista possível e, proporciona a possibilidade de torná-lo mais robusto e flexível. O agente simulador tem a seguinte ordem para sua execução:

1. registro da ontologia e da linguagem;
2. recebe o nome do XML que será decodificado;
3. recebe o ***`MRAInfo`*** e o nome das *skills* dos agentes presentes no *script*;
4. implementa o método `ReadXML()` que é subdividido em: leitura dos agentes mecatrônicos e sequencia para chamadas de *skill* (abordagem simplificada);
5. decodifica as mensagens do XML e instancia agente motor simulado que cria as máquinas de estados dos atuadores, os agentes cognitivos e o(s) agente produto.

Na sequência de execução, é necessário executar a sequência 01 até 04 que inicia com a leitura do XML e extração de todas as informações para depois poder iniciar a

```java
for (Element e : fsm.getChildren()) {
        if (e.getName().equals("States")) {
            startState = e.getAttributeValue("start");
            endState = e.getAttributeValue("end");
            for (Element e1 : e.getChildren()) {
                State state = new State(
                        e1.getAttributeValue("name"),
                        e1.getChildText("Action")
                );
                states.add(state);
            }
        } else if (e.getName().equals("Transitions")) {
            for (Element e1 : e.getChildren()) {
                Transition tran = new Transition(
                        e1.getAttributeValue("from"),
                        e1.getAttributeValue("to"),
                        e1.getAttributeValue("event")
                );
                transitions.add(tran);
            }
        } else if (e.getName().equals("DefaultTransitions")) {
            for (Element e1 : e.getChildren()) {
                Transition tran = new Transition(
                        e1.getAttributeValue("from"),
                        e1.getAttributeValue("to")
                );
                transitions.add(tran);
            }
```

Figura 25 – Decodificação do XML

instanciação dos agentes e da máquina de estados. Depois da recepção do XML e posterior execução dos *scripts* de produção, a próxima etapa consiste na visualização das interações entre os agentes do sistema, com a finalidade de executar determinada ordem de produção.

Leitura e Decodificação do XML

Para realizar a Leitura e a decodificação do XML, é necessário o uso de ferramentas disponíveis no `JADE`, mais especificamente uma biblioteca da XML, denominada SAX (*Simple API for XML*). O processamento de XML com SAX tem duas características principais: a primeira produz um laço automático que varre o documento do início ao fim; e a segunda, durante o laço automático, dispara diversos eventos para possibilitar que o desenvolvedor possa recuperar informações contidas no documento XML.

Na Figura 25, é apresentado a forma utilizada para recuperar as palavras-chave do XML. Em seguida, as etapas de decodificação, apresentadas através da comunicação, de acordo com a API SAX:

- Etapa 01: é indicado para o processador SAX o documento XML que será processado e é solicitado que ele realize a operação de *parsing* (processamento);

- Etapa 02: iniciado o *parsing*, o processador SAX iniciará a operação de *parsing*. Esta operação produz um *loop* automático que varre o documento XML do início ao fim;
- Etapa 03: durante o *loop* automático, todas as vezes que o SAX encontrar um evento relevante durante o processamento do documento, ele verificará se tem alguma ação para esse evento, o qual é relevante e acontece quando é necessário recuperar algum tipo de informação e pode ser no:
  - início do documento;
  - início de uma *tag*;
  - fechamento de uma *tag*;
  - valor entre compreendido entre uma *tag* de abertura e uma *tag* de fechamento;
  - fim do documento.
- Etapa 04: possibilidade de externar o resultado do processamento na forma de um *Script*.

A principal vantagem deste método chamado de API SAX é a sua eficiência, pois o XML não é importado para a memória. O processador SAX simplesmente implementa um laço que varre o documento do início ao fim e dispara eventos sempre que um "trecho de interesse". O SAX foi exatamente criado para fazer o processamento de XML, sem a necessidade de importar os dados para a memória. Essa etapa de decodificação do XML está contida dentro do agente simulador que fica responsável pelo encaminhamento das palavras-chave e posterior instanciamento dos agentes mecatrônicos e dos agentes produtos.

Construção da Máquina de Estados

Com o objetivo de facilitar o processamento da informação decodificada e posterior instanciamento dos agentes mecatrônicos simulados e dos agentes cognitivos do sistema dentro do agente simulador, foram criados dois métodos para realizar esta tarefa. Esses métodos são fundamentados em comportamentos do Jade e são: `InstanciaMA`, que é responsável pela instanciação dos agentes mecatrônicos decodificados do XML; e o segundo método é `SkillCall`, que é responsável pela instanciação dos agentes-produto e sua sequência de execução/produção.

O método `InstanciaMA` apresentado na Figura 26 é uma classe que encapsula as informações referentes a cada agente mecatrônico simulado que é criado e é uma classe filha de `WakerBehaviour`. Contém dados do nome do agente e seus respectivos argumentos, assim como o tempo, de acordo com as definições do desenvolvedor, que o sistema leva

```java
class InstanciaMA extends WakerBehaviour {

    private String nameAg;
    private String className;
    private String argsValues[];


    public InstanciaMA(String nameAg, String className, String[] argsValues, Agent a, long timeout) {
        super(a, timeout);
        this.nameAg = nameAg;
        this.className = className;
        this.argsValues = argsValues;
    }

    @Override
    protected void onWake() {
        System.out.println("Instancia MA: agentName=" + nameAg + "  time=" + System.nanoTime());
        AgentContainer cc = myAgent.getContainerController();

        try {
            AgentController ac = cc.createNewAgent(nameAg, className, argsValues);
            ac.start();
        } catch (StaleProxyException ex) {
            Logger.getLogger(InstanciaMA.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Figura 26 – Classe para Instanciação: `InstanciaMA`

para criar este agente. Os agentes criados são instanciados em contêineres separados e estes, por sua vez, interagem com outros agentes através de chamadas de *skill*.

O método ***SkillCall***, apresentado na Figura 27, é responsável pela definição e execução das chamas de *skill* no sistema, dada uma sequência pré-definida no XML. É uma classe que contém informações referentes ao agente que está fazendo a chamada de *skill* e seu respectivo alvo, a *skill* ou funcionalidade requisitada e os argumentos desta chamada. Nesse método é definida uma quantidade de tempo que abrange as ações de chamar a *skill*, executar esse *skill* e ter o resultado desta chamada de *skill*. Ou seja, o tempo que compreende a simulação temporal de execução das chamadas de *skill* na máquina de estados dos agentes mecatrônicos simulados.

Na construção e implementação da máquina de estados do sistema, é importante usar a definição do `FSMBehaviour` após a decodificação do XML, mais especificamente, quando ocorrer a criação do agente motor simulado, onde inicia a máquina de estados dos agentes mecatrônicos. Essa máquina descreve o funcionamento do sistema por meio de um comportamento Jade (FSM) e seus respectivos tempos para executar determinada ação quando recebem uma chamada de *skill*.

A representação da máquina de estados do módulo carimbador é apresentada na Figura 28. Quando a máquina de estados é iniciada, ela passa para um estado de espera chamado de “idle”, e fica assim até receber uma chamada de *skill* para executar a ação “stamp”. Após essa ação ser executada, decorre um período de tempo e volta ao estado

```java
class SkillCall extends WakerBehaviour {

    private String target;
    private String resultType;
    private String nameSkill;
    private String[] argTypes, argValues;

    public SkillCall(String target, String resultType, String nameSkill, String[] argTypes, String[]
        super(a, timeout);
        this.target = target;
        this.resultType = resultType;
        this.nameSkill = nameSkill;
        this.argTypes = argTypes;
        this.argValues = argValues;
    }

    @Override
    protected void onWake() {
        System.out.println("SkillCall: skillName=" + nameSkill + "  time=" + System.nanoTime());

        SkillTemplate st = new SkillTemplate(nameSkill, resultType, argTypes);
        st.setArgsValues(argValues);

        MRAInfo mraInfo = new MRAInfo();
        mraInfo.setName(target);
        mraInfo.setSkills(new SkillTemplate[]{st});

        try {
            String result = MRAServices.executeRemoteSkill(myAgent, mraInfo, st);
        } catch (SkillExecuteException ex) {
            Logger.getLogger(SkillCall.class.getName()).log(Level.SEVERE, null, ex);
        }

    }
}
```

Figura 27 – Classe para Instanciação: `SkillCall`

de espera. Essa máquina só para quando o sistema é desligado ou quando o módulo em questão é removido.

O diagrama da máquina de estados do módulo carimbador é uma representação genérica para todas os módulos, ou seja, independente da letra, a forma de funcionamento desta máquina é a mesma para todos os agentes mecatrônicos carimbo. Para execução de um estado dentro da máquina de estados, é necessária uma chamada de *skills*. Sem essa chamada não há mudança do estado de espera para a execução da ação.

A máquina de estados da esteira "conveyor" é apresentada na Figura 29. Da mesma forma que a máquina de estados do módulo carimbador, a máquina da esteira é iniciada com o sistema e aguarda no estado de espera "idle" até receber uma chamada de *skill*. A diferença em relação à máquina anterior consiste em três possibilidades: *MoveToStart*, *MoveToEnd* e *MoveTo(x,y)*. As duas primeiras possibilidades representam pontos no sistema da esteira que seriam o inicio e o fim. Quando a máquina recebe chamada para o estado MoveTo(x,y), recebe também as coordenadas referentes à posição do módulo carimbador onde serão executadas as ações.

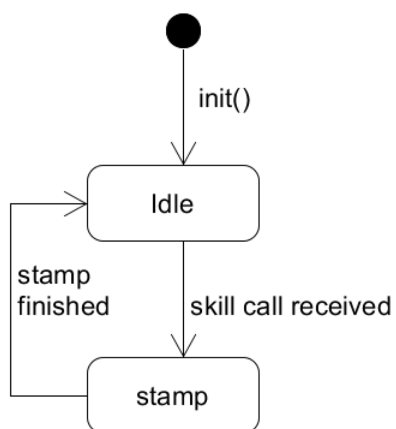


Figura 28 – Máquina de estados do módulo carimbador.

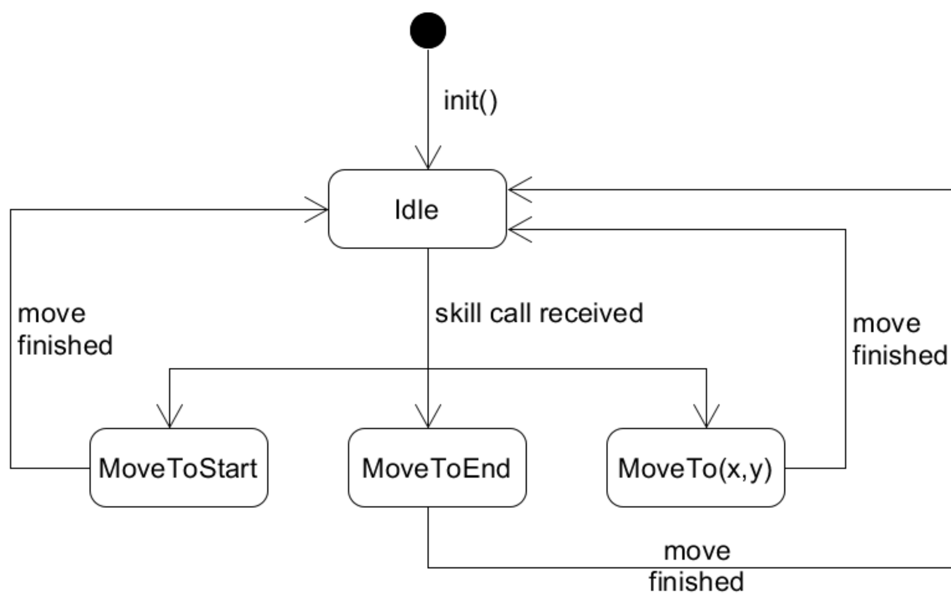


Figura 29 – Máquina de estados da Esteira.

6 Experimentos e Resultados

Neste capítulo, serão apresentados os estudos realizado, os experimentos e os resultados obtidos com base no uso e funcionamento da arquitetura EPSCore, sua aplicação e do Simulador EPS.

6.1 Comparativo das arquiteturas

Esta dissertação usa o paradigma da manufatura EPS para permitir a implementação de aplicações reais. A arquitetura, denominada EPSCore, visa estruturar a implementação de sistemas evolutivos no ambiente industrial, objetivando-se principalmente uso de agentes, flexibilização do sistema e possibilidade de auto-organização.

Por meio do paradigma da manufatura EPS, a arquitetura EPSCore é uma abordagem que traz soluções para a fabricação de produtos com alta variabilidade, produzidos em pequenos lotes, visando também ser uma abordagem simplificada no meio acadêmico. Ela é usada para o estudo e desenvolvimento de agentes inteligentes aplicados à manufatura, aprendizado de comportamentos Jade, definição de métodos para a criação de agentes, comunicação Fipa entre agentes e demais sistemas, entre outras possibilidades, tornando-a uma arquitetura didática. Fora isso, tem uma menor quantidade de agentes; nesse caso, agentes mecatrônicos e não mecatrônicos, que representam o funcionamento de um sistema.

A Tabela 4 mostra uma comparação entre estas duas arquiteturas baseadas em algumas funções dos seus agentes. EPSCore tem, por exemplo, um único MRA para o agente físico e para o agente cognitivo.

Tabela 4 – Comparação de Agentes das Arquiteturas

Características	**IADE**	**EPSCore**
Agente Físico	Agente Recurso (RA)	Agente Mecatrônico (MRA)
Agente Suporte	Agente Páginas Amarelas (YPA)	Agente Páginas Amarelas (YPA)
Agente Cognitivo	Agente Produto (PA) Agente Líder de Coalização (CLA)	Agente Mecatrônico (MRA)
Agente Integrador	-	Agente Gateway (GA)
Agente de Distribuição	Deployment Agent (DA)	-

As funcionalidades dos agentes comparados na arquitetura são parecidas, pois a arquitetura EPSCore foi inspirada na arquitetura IADE, porém apresentam algumas peculiaridades:

- *agente físico*: apresenta nomes diferentes, mas realizam basicamente a mesma função que é atuar sobre os sensores e atuadores do sistema;
- *agente suporte*: difere somente na arquitetura EPSCore, este agente não tem uma definição de área. Ou seja, comunicam-se apenas os agentes registrados neste YPA, caso queiram se comunicar com outros agentes externos, teria que ser implementado a definição de área, presente na arquitetura IADE;
- *agente cognitivo*: na arquitetura proposta, um único agente executa atividades que dois agentes executavam na arquitetura IADE;
- *agente integrador*: presente apenas na arquitetura EPSCore e tem como objetivo integrar a plataforma com outros sistemas;
- *agente distribuição*: presente apenas na IADE.

Depois de analisar a tabela, é possível verificar que a EPSCore tem, por exemplo, um único agente Mecatrônico tanto para o agente físico quanto para o agente cognitivo, o qual também encapsula informações relativas ao trasporte. Igualmente, EPSCore não possui a definição de área, a qual limita-se ao agente `YPA`; portanto, sua abordagem é mais restrita. A aplicação SIAPE foi desenvolvida explicitamente para validar a proposta da arquitetura EPSCore, ou seja, essa aplicação é exclusiva.

A arquitetura desenvolvida tem características relacionadas a sistemas mecatrônicos, simplificação da comunicação dos agentes, otimização das interações e uso de ontologia e de um simulador para reproduzir o comportamento dos agentes do sistema.

6.2 Experimentos no Simulador EPS

As atividades de produção são realizadas na arquitetura EPSCore, por meio da interação entre os agentes mecatrônicos, os quais expõem funcionalidades específicas para o sistema, na forma de chamadas de *skills*. O simulador EPS é usado para simular a aplicação SIAPE e, para isso, é necessário descrever os agentes mecatrônicos e posteriormente criar um `script` para produção dos anagramas.

Os agentes simulados presentes na aplicação são:

- `Stamper`: representa os módulos carimbadores que terão como habilidade carimbar letras no anagrama para formar determinado produto. Para uso do simulador é

criada uma classe que define uma *skill* cujo nome é "Stamp", o ***resultType*** "void" e ***argsTypes*** que especificam qual a letra será usada neste módulo;

- `Conveyor`: é a esteira que é responsável pelo transporte de produtos. No simulador é definido os três *skills* que são ***MoveToStart***, ***MoveToEnd*** e ***MoveTo***;

- `Anagran`: representa o produto do sistema. Ele é formado por uma combinação de letras disponíveis nos módulos carimbadores. No simulador, é criado um método `produce()` onde acontecem os seguintes passos de acordo com o funcionamento da aplicação:

  - definição de quem são os agentes mecatrônicos utilizados;
  - verificação da disponibilidade das letras no sistema;
  - criação de um plano de produção de acordo com as letras disponíveis;
  - execução do plano de produção.

Para realizar as comparações de performance do SIAPE e do Simulador, foi estipulada a produção dos anagramas "UFAM", "UTAM"e "UEA". Com os dados de experimentação, foram feitas comparações e algumas discussões em relação aos resultados obtidos com a finalidade de validar o Simulador EPS. Os dados do sistema físico SIAPE estão disponíveis no trabalho (AMARAL, 2016), e os dados do Simulador EPS serão apresentados em seguida e comparados com os da aplicação.

6.2.1 Criando o XML da aplicação SIAPE

Na Figura 30, é apresentado o trecho do XML que representa os agentes mecatrônicos "Stamper" que, de acordo com a descrição dos agentes do SIAPE, estão instalados no sistema e tem as letras A, U, F, T, M junto com um módulo extra (E) que está disponível para possíveis trocas. No XML é definido o *timestamp* que representa o tempo de acionamento de determinado módulo, ou seja, seu tempo de execução.

Outro trecho do XML apresenta a descrição da esteira, aqui definida como o agente *Conveyor*. O trecho referente à definição do agente `Conveyor` e seu respectivo tempo de execução é apresentado na Figura 31:

Completando o XML está a definição de chamadas de *skills*, no qual, de acordo com a descrição do funcionamento do sistema real, representam os anagramas que serão produzidos, nesse caso os anagramas "UFAM", "UTAM" e "UEA". Internamente, por meio de ferramentas disponíveis na própria arquitetura, são realizados o planejamento, a interação dos agentes e a execução da ordem de produção, de acordo com os dados presentes neste trecho do XML. Na Figura 32, é apresentada a descrição da chamada de *skills*, com o respectivo tempo para sua execução.

```xml
<MA timestamp="100" name="StamperA" class="siape.Stamper" >
    <Arg type="String">A</Arg>
    <Arg type="Integer">1</Arg>
</MA>

<MA timestamp="100" name="StamperU" class="siape.Stamper" >
    <Arg type="String">U</Arg>
    <Arg type="Integer">2</Arg>
</MA>

<MA timestamp="100" name="StamperF" class="siape.Stamper" >
    <Arg type="String">F</Arg>
    <Arg type="Integer">3</Arg>
</MA>

<MA timestamp="100" name="StamperT" class="siape.Stamper" >
    <Arg type="String">T</Arg>
    <Arg type="Integer">4</Arg>
</MA>

<MA timestamp="100" name="StamperM" class="siape.Stamper" >
    <Arg type="String">M</Arg>
    <Arg type="Integer">5</Arg>
</MA>
```

Figura 30 – Trecho do XML dos Módulos Carimbadores (*Stamper*)

```xml
<MA timestamp="100" name="Conveyor" class="siape.Conveyor" ></MA>

<MA timestamp="1000" name="Anagram1" class="siape.Anagram">
    <Arg type="String">UFAM</Arg>
</MA>

<SkillCall timestamp="2000" resultType="void" nameSkill="Produce" target="Anagram1">
</SkillCall>
```

Figura 31 – Trecho do XML da Esteira (*Conveyor*)

6.2.2 Criação da Máquina de Estados

Depois de realizada a decodificação do XML, o próximo passo é a construção a máquina de estados do sistema. Para isso, é usada a classe `FSMBehaviour` que se estende da `CompositeBehaviour`, que é uma classe abstrata para comportamentos. São definidos os estados, as transições e os eventos:

- *estados*: são definidos estados básicos de início, leitura de plano, produção e fim;
- *transições*: são as sequências definidas de um estado para outro;
- *eventos*: são as ações que acontecem em determinado estado.

As classes usadas para definir os agentes `Conveyor`, `Stamper` e `Anagran`, com seus respectivos *skills*, podem ser compreendidos como eventos disparados através de

```xml
<MA timestamp="1000" name="Anagram1" class="siape.Anagram">
    <Arg type="String">UFAM</Arg>
</MA>

<SkillCall timestamp="2000" resultType="void" nameSkill="Produce" target="Anagram1">
</SkillCall>

<MA timestamp="2000" name="Anagram2" class="siape.Anagram">
    <Arg type="String">UTAM</Arg>
</MA>

<SkillCall timestamp="3000" resultType="void" nameSkill="Produce" target="Anagram2">
</SkillCall>

<RemoveMA timestamp="6000" name="StamperM">
</RemoveMA>

<MA timestamp="10000" name="StamperE" class="siape.Stamper" >
    <Arg type="String">E</Arg>
    <Arg type="Integer">5</Arg>
</MA>

<MA timestamp="12000" name="Anagram3" class="siape.Anagram">
    <Arg type="String">UEA</Arg>
</MA>

<SkillCall timestamp="13000" resultType="void" nameSkill="Produce" target="Anagram3">
</SkillCall>
```

Figura 32 – XML da chamada de Produto (*Skillcall*)

```java
public class Stamper extends MRA {

    final Skill[] skills = new Skill []{
            new Skill(this, "Stamp", "void", new String[]{"int"}) {
                @Override
                public void execute() throws SkillExecuteException {
                    int mod = Integer.parseInt(getArgsValues()[0]);
                    System.out.println("Stamp");
                }
            }
    };
```

Figura 33 – Descrição das *skills* do `Stamper`

um `WakerBehaviour`. As *skills* dos agentes `Stamper` e `Conveyor` são apresentados na Figura 33 e na Figura 34:

6.3 Comparação entre a Simulação e a Aplicação

Na Figura 35 é apresentado o ambiente de simulação utilizado.Assim, a simulação é feita em ambiente virtual, de forma que é possível retratar as habilidades de cada módulo mecatrônico, e, se forem suficientemente básicos e gerais, um sistema de montagem evolutivo deve ser capaz de montar uma ampla gama de anagramas. Isto é, o sistema passa a ter a possibilidade de montar uma alta variabilidade de produtos e, posteriormente, testar os melhores tempos de execução e combinação de módulos. Entretanto, para métrica de simulação do tempo de produção de alguns anagramas, foi fixado o passo a passo na

```
public class Conveyor extends MRA {

    final Skill[] skills = new Skill []{
            new Skill(this, "MoveToStart", "void", new String[0]) {
                @Override
                public void execute() throws SkillExecuteException {
                  System.out.println("MoveToStart");
                }
            },
             new Skill(this, "MoveToEnd", "void", new String[0]) {
                @Override
                public void execute() throws SkillExecuteException {
                 System.out.println("MoveToEnd");
                }
            },
            new Skill(this, "MoveTo", "void", new String[]{"int", "int"}) {
                @Override
                public void execute() throws SkillExecuteException {
                    int mod = Integer.parseInt(getArgsValues()[0]);
                    int pos = Integer.parseInt(getArgsValues()[1]);
                    System.out.println("MoveTo");
                }
            }
    };
```

Figura 34 – Descrição das *skills* do `Conveyor`

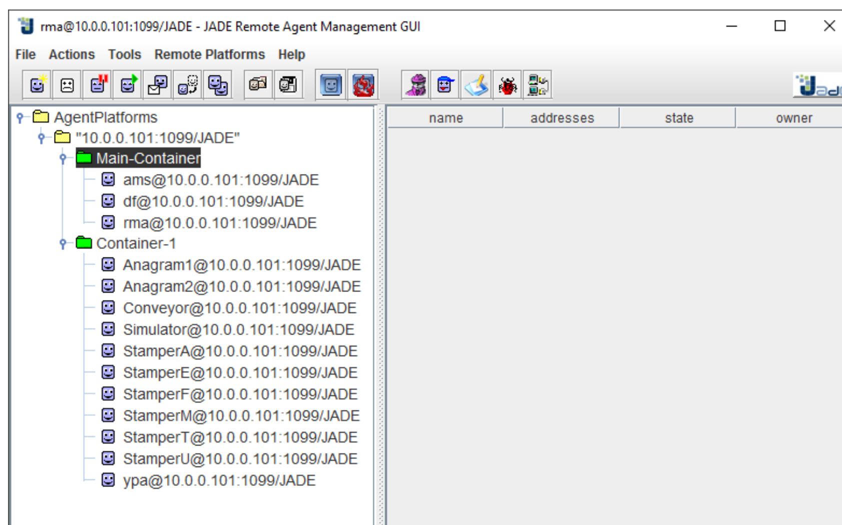


Figura 35 – Ambiente para Simulação dos Agentes

simulação para produção de "UFAM", "UTAM" e "UEA" e levado em consideração que o procedimento para simulação da fabricação é:

- verificar quais letras estão disponíveis;
- planejar a execução dos anagramas `UFAM, UTAM e UEA`;
- produzir anagrama UFAM;
- produzir anagrama UTAM;
- produzir anagrama UEA.

Tabela 5 – Comparativo do tempo de troca e acionamento dos módulos carimbadores

Testes	**SIAPE**		**Simulador EPS**	
	Acionamento Stamper	*Troca do módulo*	*Acionamento Stamper*	*Troca do módulo*
1	1,79	48,73	1,80	40
2	1,87	46,25	1,80	40
3	1,64	49,25	1,80	40
4	1,72	50,25	1,80	40
5	1,88	42,65	1,80	40
6	1,76	43,25	1,80	40
7	1,67	44,50	1,80	40
8	1,54	46,20	1,80	40
9	1,67	43,25	1,80	40
10	1,88	44,30	1,80	40
Média	***1,74***	***45,86***	***1,80***	***40***

Quanto ao tempo de execução de determinada ação, por exemplo, tempo para carimbar uma letra, a simulação temporal é feita através de comportamentos do `JADE`, mais especificamente o ***`WakeBehaviour`***. Nele são definidos quanto tempo o comportamento demora para executar determinada ação, representando o tempo de atuação do módulo mecatrônico no sistema real. Dessa forma, é fixado um tempo de execução para ações de carimbar e mover esteira, de acordo com a aplicação.

- início de produção = o método *OnWake* é definido em 0,80 segundos;
- ação de carimbar = o método *OnWake* do ***`WakeBehaviour`*** é definido em 1,80 segundos;
- mover esteira do início ao primeiro módulo = o método *OnWake* é definido em 2,5 segundos;
- mover posição no palete = o método *OnWake* é definido em 0,5 segundos;

Os demais tempos utilizados foram fixados entre 1,50 e 3,0 segundos. O tempo de execução de cada módulo mecatrônico foi detalhado no XML através do *timestamp* e posteriormente na instanciação das classes dos agentes. Dessa forma, é possível mensurar qual o tempo de produção dos anagramas desejados.

Na situação de substituição do módulo carimbador, mais especificamente na troca do módulo "T" pelo módulo auxiliar "E", são fixados os tempos para o simulador e comparados com os dados obtidos da aplicação SIAPE apresentados na Tabela 5:

O tempo médio para realizar a troca de um módulo e posterior identificação e disponibilização no plano de produção alcançou a média de 45 segundos no SIAPE, e no

Simulador foi fixado em 40 segundos. O processo de *plug and produce* é automático em ambas as situações mas, no caso da simulação, é minimizado o tempo de identificação devido à virtualização do sistema, e prejudicado, no caso da aplicação, por causa das limitações físicas.

A média temporal da sequência de operações realizadas no âmbito do simulador são:

- Simulador EPS:
  - para produzir UFAM, gastou 10,45 segundos;
  - para produzir UTAM, gastou 10,45 segundos;
  - para produzir UEA, gastou 9,8 segundos;

Para fins de comparação, foram usados os dados obtidos na aplicação SIAPE, experimento realizado pelo pesquisador (AMARAL, 2016), que obteve os seguintes tempos para produção dos mesmos anagramas:

- aplicação SIAPE - tempos de execução dos módulos carimbadores e esteira;
  - para produzir UFAM, gastou em média 10,53 segundos;
  - para produzir UTAM, gastou em média 10,63 segundos;
  - para produzir UEA, gastou em média 10,12 segundos.

Coincidentemente, no simulador EPS, foram fabricados os mesmos anagramas para poder fazer a comparação da simulação de execução dos módulos mecatrônicos. O tempo de simulação foi mais rápido porque, no Simulador, não há intervenção humana, no caso da substituição do módulo nem limitações de hardware. Os tempos definidos para acionamento dos atuadores (módulos mecatrônicos) obedecem ao critério de temporização, definidos no XML, para a execução do plano de fabricação dos anagramas na aplicação real. O tempo de troca dos módulos foi fixado em um tempo aceitável e não foi tirado uma média do tempo de troca, como acontece no SIAPE. As médias temporais encontradas no Simulador e as obtidas no SIAPE são apresentadas na Tabela 6.

O tempo de produção realizado na experimentação do Simulador e da aplicação SIAPE foi de 10 amostras temporais distintas para fabricação do mesmo anagrama. Os tempos são dados em segundos e foram comparados. Nota-se que, no simulador, por causa da fixação temporal dos módulos mecatrônicos e ausência de perdas com relação ao processo de fabricação, o tempo é o mesmo em todas as 10 amostras. Já na aplicação real, obviamente, há uma diferença temporal na maioria das amostras, por causa das limitações próprias do sistema físico.

Tabela 6 – Comparativo do Tempo de produção dos anagramas no SIAPE e no Simulador EPS

	SIAPE			**Simulador EPS**		
Testes	*UFAM*	*UTAM*	*UEA*	*UFAM*	*UTAM*	*UEA*
1	10,16	10,66	10,20	10,45	10,45	9,8
2	10,17	10,71	9,92	10,45	10,45	9,8
3	10,88	10,49	10,30	10,45	10,45	9,8
4	10,07	10,57	10,20	10,45	10,45	9,8
5	10,04	10,64	10,10	10,45	10,45	9,8
6	10,72	10,72	10,10	10,45	10,45	9,8
7	10,72	10,72	10,19	10,45	10,45	9,8
8	10,94	10,44	15,10	10,45	10,45	9,8
9	10,91	10,71	10,20	10,45	10,45	9,8
10	10,73	10,73	10,10	10,45	10,45	9,8
Média	***10,50***	***10,60***	***9,10***	***10,45***	***10,45***	***9,8***

A aplicação, quando construída no Simulador EPS, deve respeitar as restrições temporais das máquinas reais. Quanto mais próximo se colocar tais restrições no *script* de simulação (XML), mais precisa será a resposta do simulador. Nota-se que o objetivo maior do simulador é justamente a construção dos agentes mecatrônicos e a substituição do elemento físico de cada módulo, mais o seu estudo prévio do comportamento temporal do sistema como um todo.

7 Análise e Conclusão

Nesta dissertação foi desenvolvida uma arquitetura baseada no paradigma da manufatura `EPS`, denominada EPSCore. Foi também apresentada uma aplicação de um sistema mecatrônico para arquitetura e foi desenvolvido um Simulador EPS, cujo objetivo é simular o funcionamento de um sistema mecatrônico real. Dessa forma, a arquitetura EPSCore e o Simulador EPS formam juntos a principal contribuição deste trabalho.

O uso do simulador proporciona uma abordagem em que o desenvolvimento de agentes e as chamadas *skills* podem ser desenvolvidos e testados, sem necessariamente se ter a planta real. No caso específico deste trabalho, a planta real é a aplicação SIAPE.

O que é simulado é o comportamento temporal dos agentes mecatrônicos e não o comportamento físico. Por exemplo, seja um sistema que tem um carimbo que é acionado por um pistão pneumático. O simulador não está simulando as pressões internas nos êmbolos do dispositivo e sim o tempo em que as ações físicas acontecem, ou seja, são os tempos de execução de determinada ação que estão sendo simulados. Essa informação temporal está presente no *script* de simulação e está sob total controle do desenvolvedor.

Pela abordagem de implementação utilizada, o *script* de simulação é descrito como um XML. É igualmente válido lembrar que essa simulação temporal, no Simulador EPS, é feita através de comportamentos do `JADE`, mais especificamente o `WakeBehaviour`. É nele que os tempos definidos no XML são processados, ou seja, o quanto uma ação no sistema demora. Os tempos simulados, são tempos reais, isto é, a base de tempo de $1s$ na simulação corresponde a exatamente $1s$ no mundo real.

Observando-se os resultados do sistema real e da planta simulada desenvolvida sobre o EPSCore conclui-se que:

- EPSCore é uma ferramenta através da qual se pode ensinar e desenvolver agentes mecatrônicos;
- o simulador é uma máquina que reproduz o comportamento de um sistema mecatrônico sob determinadas condições. Essa simulação é realizada através de uma máquina de estados;
- o funcionamento do agente mecatrônico no simulador e na planta real são muito próximos (em termos de chamadas de *skills*), de forma que, simular uma máquina e fazer um agente conversar com essa máquina ou com uma máquina de estados, por software, não faz diferença porque estamos usando a abstração de *skills*;

- chamar *skills* em um agente mecatrônico real ou simulado, para o agente chamador é virtualmente a mesma tarefa, desde que respeitadas as limitadas temporais;

- os tempos do sistema real e do sistema simulado são equivalentes, desde que tais tempos sejam previamente conhecidos, o que é uma assunção válida, pois que tais tempos são, em geral, uma especificação de sistemas mecatrônicos, que serão construídos com tais tempos em mente. Claro que, quanto mais próximo os tempos reais dos especificados na simulação, melhor será o seu resultado;

- está-se simulando os comportamentos dos agentes mecatrônicos e da máquina real, a partir de uma máquina de estados, ou seja, está-se criando uma forma de executar *skills* dentro de uma máquina de estados;

- simulando o hardware, por meio de uma máquina de estados, recebendo chamadas de *skills*, a proposta do simulador é realizada, ou seja, pode-se desenvolver e testar o sistema de agentes mecatrônicos no simulador e substituir a máquina de estados por uma máquina real, alterando-se, para isso, apenas o código de execução dos *skills* daqueles agentes motores.

Em síntese, podemos inferir que o conjunto arquitetura e simulador atenderam à proposta deste trabalho, que foi a implementação do paradigma `EPS` aplicado à manufatura, mas com uma abordagem didática, de forma a possibilitar o aprendizado e o desenvolvimento de agentes mecatrônicos no meio acadêmico.

Podemos dizer, ainda, que o sistema apresentado possui vantagens, como a melhoria dos resultados do simulador em relação à aplicação SIAPE, disponibilização de uma ferramenta para aprendizado de agentes, entre outras vantagens. Uma desvantagem da arquitetura EPSCore é sua limitação quanto à abrangência do sistema. A implementação atual, está limitada a um sistema mecatrônico e o ideal é que o `YPA` se torne um `YPA` distribuído para ampliar sua aplicação industrial.

Propostas para trabalhos Futuros

Quanto às principais propostas para desenvolvimentos futuros relacionados aos trabalhos nesta dissertação, podemos citar:

- na arquitetura EPSCore: implementar a definição de área distribuída, em que cada máquina tem um `YPA` e estes agentes se comunicam entre si, ou seja, implementar uma abordagem de `YPA` distribuído em que cada equipamento mecatrônico se comunica com seu `YPA` e este se comunica outros `YPA's` através de um servidor. Dessa forma, um produto poderia "descobrir" todas os serviços disponíveis no sistema;

- no Simulador EPS: melhorar a simulação usando a definição de XML baseado na recomendação da `W3C - World Wide Web Consortium` (BARNETT; AKOLKAR, 2015) que especifica o uso de *State Chart XML* (`SCXML`): *State Machine Notation for Control Abstraction*, ou seja, usar o padrão de XML internacional;

- no Simulador EPS: implementar um gerador de *script* automático para, através da análise das diferentes combinações, visualizar a melhor sequência de operações para o sistema. Na proposta de trabalhos futuros, seriam inseridos apenas os requisitos do sistema, e o gerador automático de *scripts* produziria todas as combinações possíveis, diferente da forma atual, em que é realizada uma estruturação manual. Gerando todas as combinações é possível verificar/identificar a auto-organização e a emergência do sistema. Na Figura 36 é apresentada essa ideia.

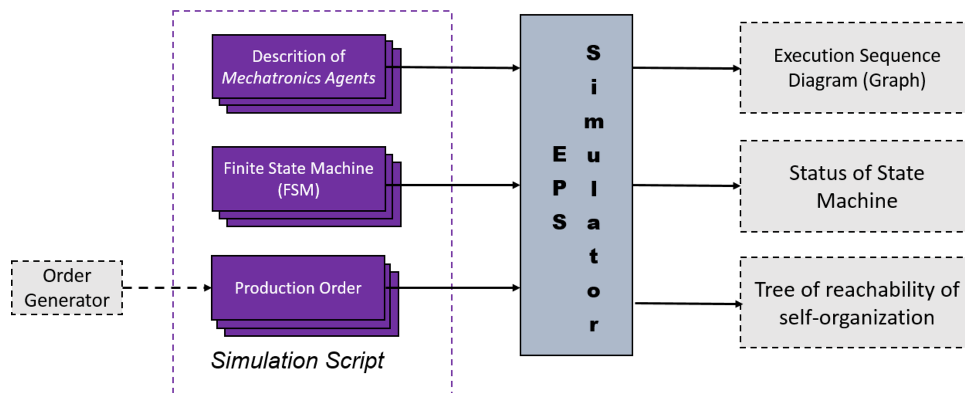


Figura 36 – Proposta do Simulador EPS para Trabalhos Futuros.

Referências

ALSTERMAN, H.; ONORI, M. Definitions, limitations and approaches of evolvable assembly system platforms. In: *Emerging Solutions for Future Manufacturing Systems*. [S.l.]: Springer, 2005. p. 367–377. Citado na página 15.

AMARAL, H. C. C. *Sistema Inteligente Ágil de Processo Evolutivo -SIAPE: um protótipo brasileiro de sistemas EPS*. Tese (Doutorado) — Faculdade de Tecnologia, Universidade Federal do Amazonas, march 2016. Citado 3 vezes nas páginas 55, 67 e 72.

BANKS, J. The future of simulation. In: *Proceedings of the 14th European Simulation Multiconference on Simulation and Modelling - Enablers for a Better Quality of Life*. [S.l.]: SCS Europe, 2000. p. 3–9. Citado na página 22.

BARATA, J.; CAMARINHA-MATOS, L.; ONORI, M. A multiagent based control approach for evolvable assembly systems. In: *2005 3rd IEEE International Conference on Industrial Informatics*. Perth, Australia: IEEE, 2005. p. 478–483. Citado 2 vezes nas páginas 10 e 31.

BARBOSA, J. et al. Enhancing adacor with biology insights towards reconfigurable manufacturing systems. In: *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*. [S.l.]: IEEE, 2011. p. 2746–2751. Citado na página 29.

BARBOSA, J. et al. Dynamic self-organization in holonic multi-agent manufacturing systems: The adacor evolution. *Computers in Industry*, v. 66, p. 99–111, 2015. Citado na página 29.

BARNETT, G. J.; AKOLKAR, I. R. State Chart XML (SCXML): State machine notation for control abstraction. 2015. Disponível em: <https://www.w3.org/TR/scxml/>. Citado na página 76.

BRUSSEL, H. V. et al. Reference architecture for holonic manufacturing systems: Prosa. *Comput. Ind.*, v. 37, n. 3, p. 255–274, nov 1998. Citado na página 14.

BURSTEIN, M. et al. OWL-S: Semantic Markup for Web Services. 2004. Disponível em: <http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/>. Citado na página 28.

BYRSKI, A. et al. Evolutionary multi-agent systems. *The Knowledge Engineering Review*, v. 30, n. 02, p. 171–186, 2015. Citado 2 vezes nas páginas 24 e 25.

CAIRE, G. JADE TUTORIAL kernel description. 2016. Disponível em: <http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf>. Citado 3 vezes nas páginas 10, 11 e 12.

CAVALCANTE, A. L. D. *Arquitetura Baseada em Agentes e Auto-organizável para a Manufatura*. Tese (Doutorado) — Escola de Engenharia, Universidade Federal do Rio Grande do Sul, 2012. Citado na página 30.

CHANDRASEKARAN, B.; JOSEPHSON, J.; BENJAMINS, V. What are ontologies, and why do we need them? *Intelligent Systems and their Applications, IEEE*, v. 14, n. 1, p. 20–26, Jan 1999. Citado na página 26.

CORBY O.; DIENG, R. . H. C. G. B. . M. G. W. E. A conceptual graph model for w3c resource description framework. In: ______. [S.l.]: Springer Berlin Heidelberg, 2000. cap. Conceptual Structures: Logical, Linguistic, and Computational Issues: 8th International Conference on Conceptual Structures, p. 468–482. Citado na página 28.

DORIGO, M.; STUTZLE, T. *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004. ISBN 0262042193. Citado na página 17.

ELMARAGHY, H. A. Flexible and reconfigurable manufacturing systems paradigms. *International Journal of Flexible Manufacturing Systems*, v. 17, n. 4, p. 261–276, 2006. Citado 2 vezes nas páginas 15 e 16.

FEIER, C. et al. Towards intelligent web services: the web service modeling ontology (wsmo). In: *2005 International Conference on Intelligent Computing (ICIC'05)*. [S.l.: s.n.], 2005. Citado na página 28.

FENSEL, D. et al. Semantic web services. In: ______. [S.l.]: Springer Berlin Heidelberg, 2011. cap. Web Service Modeling Ontology, p. 107–129. Citado na página 28.

FIPA, F. for I. P. A. FIPA Request When Interaction Protocol Specification kernel description. 2016. Disponível em: <http://www.fipa.org/specs/fipa00028/SC00028H.html>. Citado 2 vezes nas páginas 7 e 9.

FIPA, F. for I. P. A. FIPA Request When Interaction Protocol Specification kernel description. 2016. Disponível em: <http://www.fipa.org/specs/fipa00061/SC00061G.htm>. Citado na página 8.

FREI, R. et al. An architecture for self-managing evolvable assembly systems. In: *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*. [S.l.: s.n.], 2009. p. 2707–2712. Citado na página 32.

GOH MOHAN BARUWAL CHHETRI, R. K. S. K. Jade-fsm-engine: A deployment tool for flexible agent behaviours in jade. In: *International Conference on Intelligent Agent Technology*. [S.l.: s.n.], 2007. Citado na página 19.

HORROCKS I.; PATEL-SCHNEIDER, P. F. . v. H. F. From shiq and rdf to owl: the making of a web ontology language web semantics: Science. *Services and Agents on the World Wide Web*, Vol 1, n. 1, p. 7–16, 2003. Citado na página 28.

KELTON, W. D.; LAW, A. M. *Simulation modeling and analysis*. [S.l.]: McGraw Hill, 2000. Citado na página 22.

KELTON, W. D.; SADOWSKI, R. P.; SADOWSKI, D. A. *Simulation with Arena*. New York, NY, USA: McGraw-Hill, Inc., 2002. Citado na página 22.

LEITAO, P.; RESTIVO, F. Adacor: A holonic architecture for agile and adaptive manufacturing control. *Computers in industry*, v. 57, n. 2, p. 121–130, 2006. Citado 2 vezes nas páginas 29 e 31.

MEHRABI, M. G.; ULSOY, A. G.; KOREN, Y. Reconfigurable manufacturing systems: Key to future manufacturing. *Journal of Intelligent Manufacturing*, v. 11, n. 4, p. 403–419, 1999. Citado 2 vezes nas páginas 1 e 15.

MUSTAFEE, N. et al. Hybrid simulation studies and hybrid simulation systems: Definitions, challenges, and benefits. In: *2015 Winter Simulation Conference (WSC)*. [S.l.: s.n.], 2015. p. 1678–1692. Citado na página 34.

NEVES, P. et al. Exploring reconfiguration alternatives in self-organising evolvable production systems through simulation. In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. [S.l.]: IEEE, 2014. p. 511–518. Citado na página 34.

ONORI, M. Evolvable assembly systems: A new paradigm? In: *33rd International Symposium on Robotics*. [S.l.: s.n.], 2002. Citado na página 15.

ONORI, M.; BARATA, J. Evolvable production systems: New domains within mechatronic production equipment. In: *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*. [S.l.]: IEEE, 2010. p. 2653–2657. Citado na página 16.

ONORI, M.; BARATA, J.; FREI, R. Evolvable assembly systems basic principles. In: *Information Technology For Balanced Manufacturing Systems*. [S.l.]: Springer US, 2006. p. 317–328. Citado na página 32.

ONORI, M. et al. The ideas project: Plug and produce at shop-floor level. *Assembly Automation*, v. 32, p. 124–134, 2012. Citado 2 vezes nas páginas 10 e 29.

PEGDEN, C. D.; SADOWSKI, R. P.; SHANNON, R. E. *Introduction to Simulation Using SIMAN*. [S.l.]: McGraw-Hill, 1995. Citado 2 vezes nas páginas 22 e 23.

PEIXOTO, J. A. *Desenvolvimento de Sistemas de Automação da Manufatura usando Arquiteturas Orientadas a Serviços e Sistemas Multi-Agentes*. Tese (Doutorado) — Escola de Engenharia, Universidade Federal do Rio Grande do Sul, 2012. Citado 2 vezes nas páginas 13 e 30.

RAHATULAIN, A.; QURESHI, T. N.; ONORI, M. Modeling and simulation of evolvable production systems using simulink/simevents. In: *ECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*. [S.l.]: IEE, 2014. p. 2591–2596. Citado na página 34.

RIBEIRO, L. et al. Evolvable production systems: An integrated view on recent developments. In: HUANG, G.; MAK, K.; MAROPOULOS, P. (Ed.). *Proceedings of the 6th CIRP-Sponsored International Conference on Digital Enterprise Technology*. [S.l.]: Springer Berlin Heidelberg, 2010. p. 841–854. Citado 2 vezes nas páginas 16 e 32.

RIBEIRO, L.; BARATA, J.; FERREIRA, J. Emergent diagnosis for evolvable production systems. In: *2010 IEEE International Symposium on Industrial Electronics*. [S.l.]: IEEE, 2010. p. 2647–2652. Citado na página 34.

RIBEIRO, L.; BARATA, J.; MENDES, P. Mas and soa: complementary automation paradigms. In: *Innovation in manufacturing networks*. [S.l.]: Springer US, 2008. p. 259–268. Citado na página 24.

ROSA, R. P. G. *Assessing Self-Organization and Emergence in Evolvable Assembly Systems (EAS)*. Tese (Doutorado) — Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Set 2013. Citado na página 30.

SCHRIBER, T. J. A quick gpss quiz. *SIGSIM Simul. Dig.*, v. 6, n. 1, p. 32–32, 1974. Citado na página 22.

SETHI, A. K. S. . S. P. Flexibility in manufacturing:a survey. *International Journal of Flexible Manufacturing Systems*, Vol 2, n. 2, p. 289–328, july 1990. Citado na página 13.

SILVA, L. A. de Morais e. *Estudo e Desenvolvimento de Sistemas Multiagentes usando JADE: Java Agent Development Framework*. Tese (Doutorado) — Centro de Ciências Tecnológicas, Universidade de Fortaleza, 2003. Citado na página 7.

TAMMA, V. et al. An ontology based approach to automated negotiation. In: PADGET, J. et al. (Ed.). *Agent-Mediated Electronic Commerce IV. Designing Mechanisms and Systems*. [S.l.]: Springer Berlin Heidelberg, 2002. p. 219–237. Citado na página 26.

UEDA, K. A concept for bionic manufacturing systems based on dna-type information. In: *Proceedings of the IFIP TC5 / WG5.3 Eight International PROLAMAT Conference on Human Aspects in Computer Integrated Manufacturing*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1992. p. 853–863. Citado na página 13.

UEDA, K. Emergent synthesis approaches to biological manufacturing systems. In: CUNHA, P.; MAROPOULOS, P. (Ed.). *Digital Enterprise Technology*. [S.l.]: Springer US, 2007. p. 25–34. Citado na página 14.

UEDA, K.; VAARIO, J.; FUJII, N. Interactive manufacturing: Human aspects for biological manufacturing systems. *CIRP annals-Manufacturing technology*, v. 47, n. 1, p. 389–392, 1998. Citado na página 28.

VALCKENAERS, P.; BRUSSEL, H. V. Holonic manufacturing execution systems. *CIRP Annals-Manufacturing Technology*, v. 54, n. 1, p. 427–432, 2005. Citado na página 29.

Wikipedia, t. f. e. Software framework. 2016. Disponível em: <https://en.wikipedia.org/wiki/Software_framework>. Citado na página 21.

WOLF, T.; HOLVOET, T. Engineering self-organising systems: Methodologies and applications. In: ______. [S.l.]: Springer Berlin Heidelberg, 2005. cap. Emergence Versus Self-Organisation: Different Concepts but Promising When Combined, p. 1–15. Citado 2 vezes nas páginas 17 e 18.

WOOLDRIDGE, M. Reactive and hybrid agents. *An Introduction to MultiAgent Systems*, p. 89–101, 2002. Citado na página 6.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. *Knowledge engineering review*, v. 10, n. 2, p. 115–152, 1995. Citado na página 5.