DOI: 10.48011/asba.v2i1.1492

Descrição de um processo de manufatura utilizando arquitetura baseada em Sistema Multiagente

Fábio Ricardo Rocha dos Santos *
Rafael da Silva Mendonça **
Renan Landau Paiva Medeiros ***
André Luiz Duarte Cavalcante ****

* *Faculdade de Tecnologia, Universidade Federal do Amazonas, AM, (e-mail: fabioricardo@ufam.edu.br).*
** *Faculdade de Tecnologia, Universidade Federal do Amazonas, AM, (e-mail: mendoncarms@ufam.edu.br).*
*** *Faculdade de Tecnologia, Universidade Federal do Amazonas, AM, (e-mail: renanlandau@ufam.edu.br).*
**** *Faculdade de Tecnologia, Universidade Federal do Amazonas, AM, (e-mail: andrecavalcante@ufam.edu.br).*

Abstract: A common problem in modern manufacturing systems is the recurring need for configuration of assembly lines due to product changes, especially due to the so-called "mass customization". The challenge, therefore, is the rapid reconfiguration to allow the production of several customized products. For this reason, several paradigms for manufacturing have been proposed, and among them are those that bring the concept of evolvable systems (EPS/EAS), which are capable of self-organization in the industrial environment. The present work is a study on a multi-agent architecture based on the Evolvable Production Systems (EPS) paradigm, which develops the concept of cyber-physical agent, allowing the possibility of self-organization and emergency studies. The architecture is called EPSCore and is used as a basis for the study and description of the proposed system.

Resumo: Um problema comum em sistemas de manufatura modernos é a recorrente necessidade de configuração das linhas montagem em função da mudança do produto, especialmente devido à chamada "costumização em massa". O desafio, portanto, é a rápida reconfiguração para permitir a produção de vários produtos customizados. Para isso, diversos paradigmas para a manufatura têm sido propostos, e, entre eles, estão os que trazem o conceito de sistemas evolutivos (EPS/EAS), os quais são capazes de inserir auto-organização no ambiente industrial. O presente trabalho é um estudo sobre uma arquitetura multiagente baseada no paradigma de Sistemas Evolutivos de Produção (EPS), a qual desenvolve o conceito de agente cyber-físico, permitindo a possibilidade de estudo de auto-organização e emergência. A arquitetura é denominada EPSCore e é utilizada como base para o estudo e descrição do sistema proposto.

Keywords: Agent, Cyber Physical Systems; Evolvable Assembly/Production Systems; Multi-Agent Systems; Manufacturing.
Palavras-chaves: Agentes, Sistemas Cyber-físicos; Sistemas Evolutivos de Produção/Montagem; Sistemas Multiagentes; Manufatura.

1. INTRODUÇÃO

Uma das principais preocupações de qualquer sistema de manufatura é a responsividade ao mercado, e essa pode ser obtida projetando sistemas de manufatura capazes de serem atualizados facilmente (Koren et al., 2018). Tal capacidade é um fator que pode prover uma vantagem competitiva (Suzić et al., 2018) e, por isso, um dos desafios com os quais os sistemas de produção modernos têm de lidar é a necessidade da criação de sistemas de montagem capazes de realizar autoadaptação e auto-organização (Cavalcante, 2012). O cenário ideal, e portanto desejado, é que o tempo de reconfiguração do sistema de produção, para que o mesmo esteja apto a um novo produto, seja próximo a zero, aumentando a produtividade.

Assim, surgem propostas em meio à necessidade de sistemas de montagem capazes de evoluir continuamente em resposta às mudanças nos requisitos e demanda do produto (Sanderson et al., 2019). Tais soluções buscam a fabricação ágil de produtos, os quais devem ter alta variabilidade dentro de pequenos lotes de produção; a chamada customização em massa. Dentre essas propostas, destacam-se os

⋆ A Universidade Federal do Amazonas promoveu suporte financeiro a este trabalho.

paradigmas EPS e EAS (Evolvable Production/Assembly System), os quais propõem um sistema evolutivo baseado em módulos reconfiguráveis com tarefas específicas (Alsterman and Onori, 2004), permitindo a produção focada na agilidade em meio à modificação dos módulos (meio de produção).

Este trabalho traz uma aplicação (através de um protótipo de manufatura) da arquitetura EPSCore, que é uma modelagem baseada em EPS capaz de reconhecer os módulos que estão ativos em um determinado momento e se auto-organizar, de modo a formar a sociedade de agentes necessária para a execução de algum objetivo no sistema (Mendonça, 2016) em tempo de execução. A arquitetura possui a característica chamada *plug and produce*, em que cada módulo pode ser retirado do, ou colocado no, sistema como parte integrante de seu funcionamento normal, permitindo, assim, uma flexibilidade não descrita inicialmente de maneira explícita.

2. FUNDAMENTAÇÃO TEÓRICA

2.1 Agentes

A palavra agente é, intuitiva e linguisticamente, definida como aquele que age. Apesar de não haver uma só definição, na computação, é um consenso que um agente é uma entidade computacional que possui autonomia, características de um sistema arbitrário e/ou se comporta como um agente humano, trabalhando para alguns clientes em busca de sua própria agenda (Bellifemine et al., 2007).

Assim, um agente pode ser definido como uma entidade de software que pode perceber e/ou atuar em seu ambiente. Um agente racional ou inteligente é, portanto, um agente capaz de tomar uma decisão sobre como atuar, a partir do que percebe do seu meio.

Para este trabalho, outro conceito muito importante a ser estabelecido é o de agente cyber-físico, mas para isso, conceitos como o de módulo e entidade cyber-física precisam ser esclarecidos. Um módulo é uma entidade autocontida que possui uma função específica, uma interface bem definida e a possibilidade de interagir com outros módulos (Cavalcante, 2012). Já uma entidade cyber-física é uma parte de um sistema cyber-físico (CPS) que, por sua vez, é definido como a integração do processo físico de controle e monitoramento computadorizado em rede (Lee, 2007). Sendo assim, uma entidade cyber-física é aquela que integra seu hardware, permitindo, a esse, funcionar com uma representação cibernética, atuando como uma representação virtual da parte física (Leitão et al., 2016). Por fim, um agente cyber-físico pode ser definido como um agente que integra o módulo ao sistema (rede), podendo receber e enviar informações ao hardware.

Quando um conjunto de agentes do mesmo tipo ou de tipos diferentes são implementados de forma que, conjuntamente, resolvam um problema, temos um sistema Multiagentes (Multi-Agent Systems - MAS). Um MAS é definido simplesmente como um sistema que compreende dois ou mais agentes ou agentes inteligentes (McArthur et al., 2007).

Comunicação entre Agentes: Protocolos FIPA A fim de permitir a interoperabilidade entre agentes físicos, independente da linguagem de programação utilizada para o seu desenvolvimento, a Foundation for Intelligent Physical Agents (FIPA) definiu vários protocolos e regras para comunicação entre agentes em vários níveis: aplicação, arquitetura, comunicação, gerenciamento de agentes e transporte de mensagens. Dentre esses, é especialmente importante, para esse trabalho, citar a Agent Communication Language specifications (ACL) e o FIPA Contract Net Interaction Protocol.

As especificações FIPA ACL representam um conjunto de normas que se destinam à padronização da comunicação propriamente dita dos agentes. Uma mensagem FIPA ACL contém um conjunto parâmetros associados às propriedades que auxiliam na comunicação.

O FIPA Contract Net Interaction Protocol é usado na comunicação entre agentes do tipo $1 \times n$ em que um agente demonstra interesse em certo serviço mandando uma mensagem (Call For Propose - `CFP`) para os agentes que o oferecem. Assim, tais agentes podem retornar mensagens (`Propose`) com suas propostas ou simplesmente recusar (`REFUSE`). Em seguida, o agente iniciador dessa comunicação pode escolher com quais agentes o serviço será feito.

JADE – Java Agent DEvelopment Java Agent DEvelopment Framework (JADE) é um framework para o desenvolvimento de aplicações multiagentes em Java seguindo as especificações da FIPA. A escolha do framework para o projeto é justificada, além da coerência com a FIPA, pelo fato da arquitetura EPSCore ter sido desenvolvida utilizando o mesmo.

Ciclo de Vida de um Agente No JADE, um agente é modelado através de uma classe Java chamada `Agent`, a qual implementa uma *thread* Java que executa comportamentos continuamente. O ciclo de vida de um agente é definido como uma máquina de estados: após ser iniciado, o agente vai para o estado ativo e ali permanece até ser explicitamente colocado em espera ou suspenso (bloqueado).

Comportamentos Durante o tempo de vida do agente, esse irá executar seus comportamentos. Um comportamento é uma entidade de execução dentro do agente, modelada através da classe `Behaviour`. Um agente pode ter vários comportamentos e esses são normalmente adicionados ao agente no método `setup()`, mas podem ser adicionados de qualquer lugar do agente, inclusive a partir de outros comportamentos.

O comportamento global do agente, isto é, o comportamento resultante final, será o resultado das ações de todos os comportamentos do agente. O JADE define alguns comportamentos padrões que estão disponíveis e têm funcionalidades próprias. Para este trabalho, são especialmente importantes:

- `OneShotBehaviour`: permite ao programador implementar comportamentos que serão executados somente uma vez;
- `SequentialBehaviour`: agenda no JADE a execução de subcomportamentos de modo sequencial e termina quando todos são executados;

- `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.

2.2 Conceitos de Auto-organização e Emergência

Emergência e auto-organização referem-se a dois fenômenos completamente distindos. Por isso, essa seção tem por objetivo esclarecer tais conceitos.

O conceito de auto-organização é bem intuitivo: um processo dinâmico e adaptável, em que os sistemas adquirem e mantêm estruturas sozinhos, sem controle externo, onde essa 'estrutura' pode ser uma estrutura espacial, temporal ou funcional (De Wolf and Holvoet, 2004). Aplicado a sistemas de manufatura, para esse trabalho, destaca-se a organização automática, sem controle externo, do sistema de produção ao ser modificado (inclusão ou remoção de um módulo), para uma pronta produção.

Por outro lado, um sistema exibe emergência quando há emergentes coerentes no nível macro que dinamicamente aparecem das interações das partes no nível micro. Tais emergentes são uma novidade radical com respeito às partes individuais do sistema (De Wolf and Holvoet, 2004). Em outras palavras, emergência é a característica que permite um comportamento global (do sistema), não antes programado explicitamente, a partir de comportamentos/fênomenos individuais (partes que constituem o sistema). Exemplos simples de emergência são: caminhos globais de feromônios que surgem dos caminhos locais desses feromônios, o movimento de enxame de um bando de pássaros, um engarrafamento devido às interações dos carros e etc. (De Wolf and Holvoet, 2004).

2.3 Sistemas Evolutivos: EPS/EAS

Um sistema evolutivo pode ser definido como um sistema dinâmico e auto-organizado, que consegue modificar a sua estrutura devido às mudanças ambientais relevantes. Como dito na seção anterior, uma mudança especialmente relevante é a inserção, remoção ou reposição de módulos ao sistema, o que caracteriza a plugabilidade, a qual é parte do funcionamento normal do sistema. Esse conceito concorda com a definição para EAS/EPS de Onori et al. (2006).

Ainda para o autor, "EPS é baseado em muitos elementos (módulos do sistema) simples, específicos à tarefa e reconfiguráveis, que permitem evolução contínua do sistema de montagem". Já EAS é "um sistema de montagem que pode coevoluir conjuntamente com o produto e processo de montagem" (Frei et al., 2009).

Em outras palavras, EAS e EPS são conceitos similares, exceto pelo nível que são considerados, pois EAS está focado no nível do dispositivo, enquanto EPS está focado no nível da fábrica. Portanto, no nível da célula, dependendo das circunstâncias, um mesmo sistema pode ser chamado tanto de EAS quanto de EPS (Cavalcante, 2012). Por isso, nesse trabalho, os termos serão usados como sinônimos.

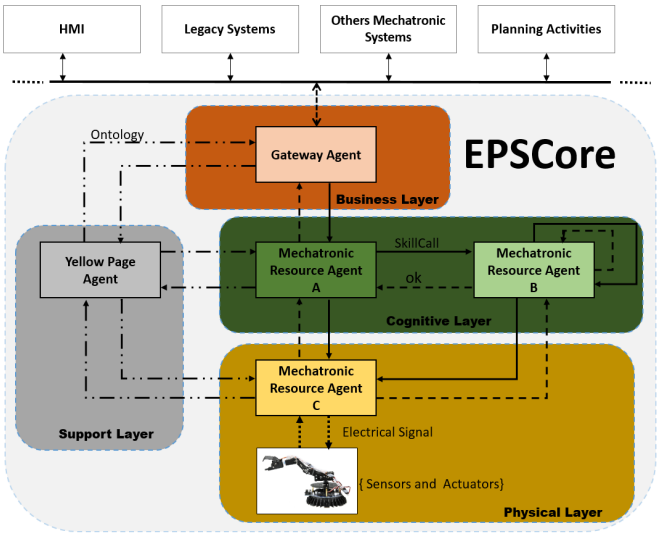

Figura 1. Arquitetura EPSCore. Fonte: (Mendonça, 2016).

3. ARQUITETURA EPSCORE

A arquitetura EPSCore (Figura 1) foi inicialmente proposta por (Mendonça et al., 2016), posteriormente descrita em (Mendonça, 2016) e é inspirada na arquitetura IADE, que é a arquitetura de um projeto da União Européia de mesmo propósito. A arquitetura EPSCore é fundamentada basicamente em MAS e EPS, de forma que a inteligência do sistema está distribuído entre os agentes. Em outras palavras, a confecção do produto acontece depois das interações entre os agentes do sistema (Mendonça, 2016).

A divisão em camadas existe como abordagem simplificada para facilitar a solução de problemas mais complexos partindo do princípio de "dividir para conquistar". De acordo com Mendonça (2016), são elas:

- Camada física: é formada por agentes cyber-físicos 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 cyber-físicos que contêm inteligência capaz de coordenar outros agentes cyber-físicos através da troca de mensagens e chamadas de skills que acontecem no sistema;
- Camada de suporte: é formada por agentes não cyber-físicos que dão suporte à execução de serviços e descoberta de funcionalidades no sistema para os demais agentes;
- Camada de negócios: contém agentes não cyber-físicos que representam a ponte de comunicação entre a arquitetura EPSCore e os outros sistemas que podem ser desde um monitor a um outro sistema cyber-físico.

3.1 Conceitos Importantes

Como dito anteriormente, a implementação da arquitetura é feita em Java, com o framework JADE. Para conseguir uma comunicação eficaz e concisa, a arquitetura conta com classes que estabelecem conceitos dentro de uma ontologia própria da mesma. A ontologia define uma estrutura comum de significados para serem usados pelos agentes na comunicação sem que aconteça a possibilidade de ambi-

guidades. Assim, é criado um vocabulário semântico para a troca de mensagens da aplicação. No que diz respeito à ontologia, segundo Mendonça (2016), temos as seguintes classes:

- `EPSOntology`: é um `BeanOntology` (instância da classe nativa do Java para ontologia) responsável pela definição, registro e instanciação da ontologia;
- `Register`: é responsável pelo registro dos agentes cyber-físicos no agente de páginas amarelas através de informações contidas no objeto `MRAInfo` do mesmo;
- `Unregister`: é um `AgentAction` (pois herda dessa classe do JADE que representa uma ação de um agente) encarregado de cancelar o registro dos agentes cyber-físicos dentro do agente páginas amarelas;
- `Execute`: é um `AgentAction` (pois herda dessa classe) responsável pela chamada e execução dos skills;
- `Search`: disponibiliza uma busca das skills (funcionalidades) dos agentes registrados no sistema.
- `GetAllMRAInfo`: responsável por realizar uma solicitação de todos os `MRAInfo` cadastrados.

Além disso, é importante salientar o conceito de skill, pois o mesmo permeia todo o funcionamento da arquitetura. A arquitetura conta com 3 classes para estabelecer esse conceito e ainda segundo Mendonça (2016), são elas:

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

Outras classes são fundamentais na arquitetura, principalmente descrevendo os agentes. Elas são usadas como base para o desenvolvedor definir (descrever) exatamente como seu sistema deverá funcionar. São elas:

- `MRA`: define um agente cyber-físico. Logo, para se criar um agente cyber-físico é necessário que a classe em questão seja filha de `MRA`;
- `YPA`: classe que define o agente de páginas amarelas. Interage com todas as outras camadas da arquitetura EPSCore e oferece serviços como: registro, pesquisa e cancelamento de registro;
- `MRAInfo`: classe que define informações específicas sobre os agentes cyber-físicos;
- `YPAServices`: classe que expõe métodos que encapsulam os serviços do agente de páginas amarelas. Logo, ao solicitar um serviço para tal agente, o agente solicitante usará os métodos abstratos dessa classe;
- `Product`: define um agente que representa um produto genérico. Implementa o método `produce()`, que é um plano de produção, isto é, um conjunto de comportamentos que, quando executados, faz o agente desempenhar seu papel no sistema.

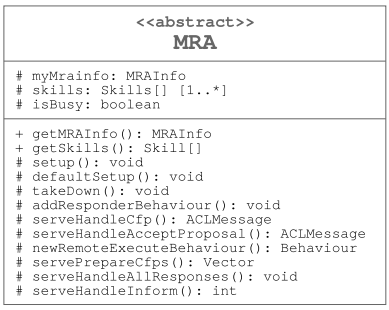

Figura 2. Classe MRA ajustada.

4. AJUSTES NA ARQUITETURA

Ao longo da descrição do Staudinger, alguns ajustes nas classes-base da arquitetura foram feitas de modo a facilitar o trabalho. Esta seção comentará tais feitos.

4.1 Rede de Contratos e Execução Remota

A fim de evitar ter de implementar um método para responder outros agentes em cada agente cyber-físico, decidiu-se estabelecer um método padrão na classe base desses agentes (isto é, na classe `MRA`) no qual adiciona-se um comportamento respondedor. Assim, aproveitou-se para estabelecer um alinhamento com o protocolo FIPA Contract Net Interaction, uma vez que sistemas manufatura podem ter mais de um módulo mecatrônico que desempenham a mesma função, podendo refletir em mais de um agente cyber-físico disponibilizando a mesma skill. Tal método é o `addResponderBehaviour()` e, na prática, ele implementa o papel de um "participante"no protocolo citado. A Figura 2 ilustra a classe `MRA` ajustada.

Para completar a rede de contratos, o método `newRemoteExecuteBehaviour()`, que recebe somente uma `skill` como parâmetro, foi criado para implementar o papel de um "iniciador"no protocolo citado. Logo, tal método permite ao agente iniciar uma rede contrato para executar uma dada skill. Além disso, o método encapsula a solicitação de busca feita para o `YPA`. A Figura 3 ilustra a comunicação desencadeada ao ser feita uma execução remota de uma determinada skill.

4.2 Plano de Produção

Os agentes do tipo `Product` desempenham seus papéis no sistema através da execução de seus respectivos processos (plano de produção), os quais nada mais são que um conjunto de chamadas remotas de skills ou de outros Agentes.

Na descrição, esses agentes fazem chamadas remotas de skills de agentes da camada física. E, a fim de organizar tal conjunto de chamadas, foram criadas classes que implementam um plano de produção na arquitetura. São elas:

- `Item`: é uma interface que tem somente um método: `execute()`. Tal método deve ser implementado de forma a retornar um comportamento que realiza a execução de um item num plano de produção;

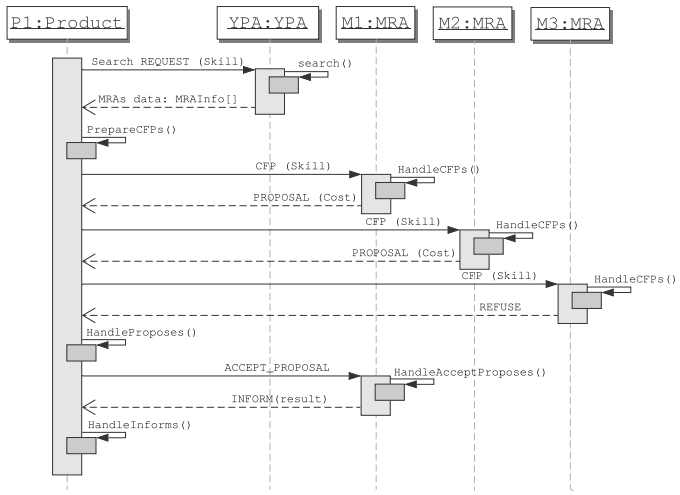

Figura 3. Comunicação em uma execução remota.

- `PlanItem`: classe que implementa a interface `Item` e define um item simples dentro de um plano de produção. Tem um atributo do tipo `skill`, o qual define a skill a ser executada, e um atributo do tipo `MRA`, o qual define o agente pelo qual o item será executado (isto é, o dono do plano no qual o item está inserido). O método `execute()` é definido como a execução remota da skill citada por parte do agente também citado;
- `DecisionItem`: classe que implementa a interface `Item` e define um item de decisão dentro de um plano de produção através de 3 objetos do tipo `PlanItem`: um para decisão e dois para escolhas. No método `execute()` cria e retorna-se um objeto do tipo `FSMBehaviour`, no qual a execução da decisão é o estado inicial e as execuções das escolhas são os estados finais. Caso o resultado da decisão seja 0, a escolha 0 (`choice0`) se segue. Caso seja 1, a escolha 1 (`choice1`) é escolhida;
- `Plan`: define um plano de produção no sistema e provém métodos que encapsulam todo o processo de produção para serem usados finalmente na classe `Product`. Tal classe tem um atributo do tipo `Plan` e esse plano é criado utilizando os métodos de adicionar itens (`addNewPlanItem()` e `addNewDecisionItem()`). Ao serem criados, tais itens são guardados num `ArrayList` e, uma vez que o plano seja executado, cada item tem seu comportamento de execução extraído e guardado num `SequentialBehaviour`, o qual é adicionado ao agente dono do plano (`Owner`).

5. HARDWARE DO SISTEMA

O conjunto Staudinger GMBT é um protótipo de um sistema de manufatura que tem como objetivo principal simular uma linha de produção industrial automática, utilizando esteiras, sensores e atuadores para confeccionar "produtos"e armazená-los, com uso da intervenção humana somente em seu controle por meio de um controlador lógico programável (CLP).

O conjunto é formado por módulos que cumprem funções específicas. São eles:

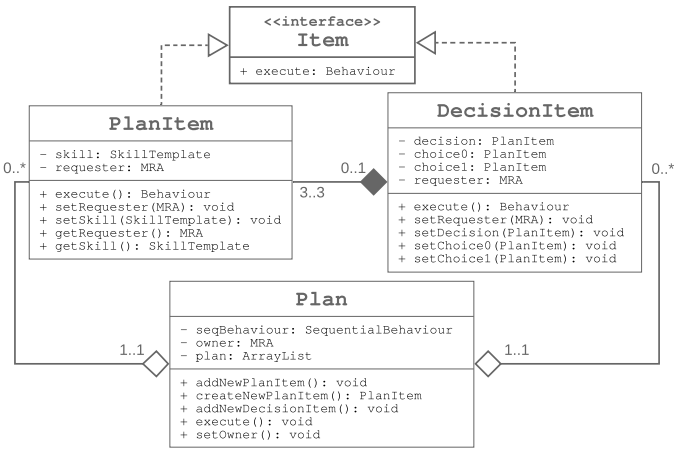

Figura 4. Classes que implementam um plano de produção na arquitetura para a descrição.

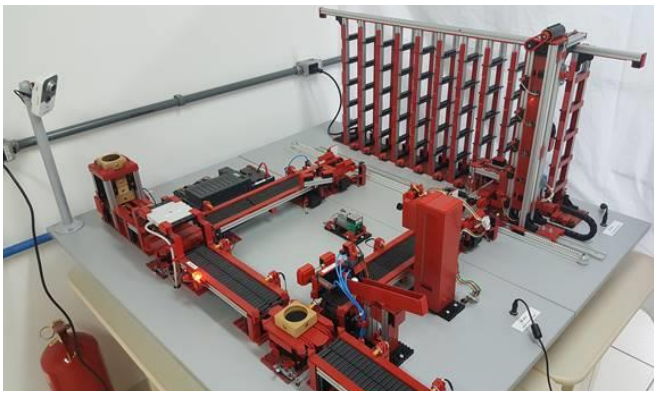

Figura 5. Conjuto Staudinger GMBT.

- Resgister Storage with Conveyor Belt: é responsável por ser o começo da produção. Para tal, possui uma pilha de caixotes, da qual os mesmos são retirados para serem inseridos na linha de produção;
- Rotate Conveyor Belt: é composto por esteiras rotativas. Possui a função de fornecer direções diferentes aos caixotes. Para tal, avalia a cor do caixote (por meio de seus sensores) e o redireciona à esteira (próximo módulo) adequada.
- High Level Storage Warehouse: Este módulo é composto por uma esteira transportadora, outras duas de acesso, elevador e armazém para as peças. A função deste modulo é simular um processo de armazenagem de produto, algo bem comum na indústria;
- Conveyor Belt with Machine Tool: é composto por uma esteira transportadora e um atuador capaz de subir e abaixar uma ferramenta. A função deste conjunto é simular algum tipo de mecanismo existente na indústria, como uma prensa ou um cortador;
- Pneumatic picking: é composto por uma esteira transportadora e um atuador pneumático capaz de inserir pequenas esferas nos caixotes. A função deste modulo é simular a inserção de algum recurso existente na indústria, como, por exemplo, bombons em uma caixa de chocolates.

Por ser controlado por meio de um controlador lógico programável (CLP), o Staudinger admite uma série de possibilidades de comportamentos, desde que o progra-

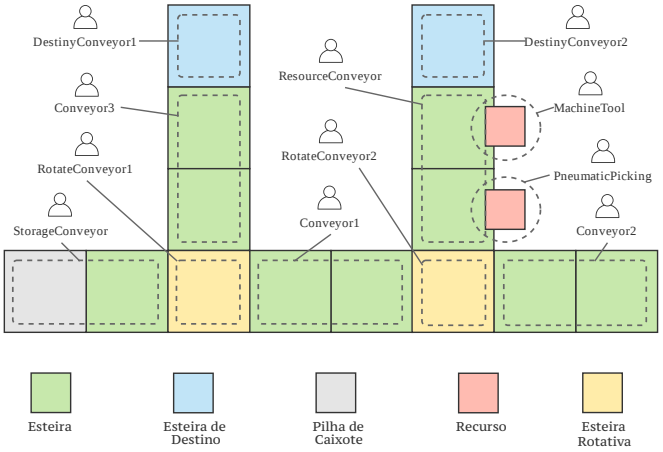

Figura 6. Agentes cyber-físicos da camada física e suas respectivas áreas de atuação no hardware.

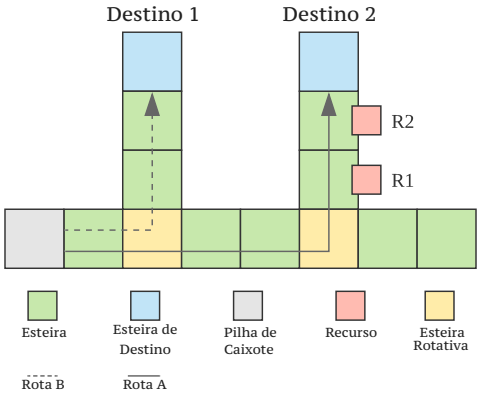

Figura 7. Rotas para os caixotes no comportamento adotado.

mador os implementem. Tal fato permite também que o Staudinger possa disponibilizar microsserviços, como, por exemplo, o de mover uma determinada esteira por um determinado tempo. Com isso, o funcionamento geral pode ser fragmentado em pequenos serviços que poderão ser executados sempre que um outro sistema (um sistema multiagente, no caso deste trabalho) os solicitem através de uma rede. É a partir disso que a arquitetura EPSCore poderá ser aplicada.

5.1 Rotina de Funcionamento

Para este trabalho, o comportamento adotado para o Staudinger ignora a fase de armazenagem, uma vez que algumas esteiras do armazém se encontram defeituosas. Assim, os destinos finais dos caixotes são as esteiras de acesso ao armazém (esteiras de destino), que são duas.

O usuário final poderá requisitar uma série de produtos de uma vez. Tais produtos são caixotes que podem ser da cor preta ou verde e podem ou não conter esferas (bolinhas). Ao solicitar uma produção, o usuário especificará a cor do caixote e a quantidade de bolinhas desejada.

Para cada pedido dessa lista de produtos, o sistema irá pegar um novo caixote da pilha e verificar se sua cor coincide com a do pedido através dos sensores da primeira Rotate Conveyor (esteira rotativa), que se encontra próxima à pilha de caixote. Se sim, tal caixote é usado na produção (rota A), onde poderá receber a quantidade de bolinhas solicitada e ser tampado. Se não, tal caixote é levado por outro caminho (rota B) para que seja inserido no armazém, para que possa ser usado futuramente. Uma vez que a fase de armazenagem foi deixada de lado, a rota B acaba na esteira de destino 1, enquanto a rota A acaba na esteira de destino 2. A Figura 7 ilustra tais rotas.

6. APLICAÇÃO STAUDINGER

6.1 Classes dos agentes

Como dito anteriormente, a arquitetura proporciona várias classes que servem de base para uma modelagem MAS. Cabe ao aplicador, então, usar tais classes a sua disposição, assim como criar novas classes (específicas para aplicação) que as utilizem. Para a descrição do Staudinger, na camada física, tem-se as seguintes classes:

- `Conveyor`: classe filha de `MRA` que modela os módulos Conveyor Belt e tem a skill `move`, a qual move o caixote para a posição solicitada ligando o motor da esteira;
- `StorageConveyor`: classe filha de `MRA` que modela o módulo Register Storage with Conveyor Belt e, para isso, conta com a skill `getNewBox`, que verifica se tem caixote na pilha do módulo e, caso tenha, o move através da esteira para o próximo módulo;
- `RotateConveyor`: classe filha de `MRA` que modela os módulos Rotate Conveyor Belt e conta com três skills: `receive`, `checkColor` e `move`. A primeira é responsável por rotacionar a esteira até a posição solicitada e ligar o motor, de modo a receber um caixote dessa direção. A segunda verifica se a cor do caixote coincide com a cor passada como parâmetro, se sim, retorna `true`. A última move o caixote para a posição solicitada utilizando a esteira do módulo;
- `ResourceConveyor`: classe filha de `MRA` responsável por modelar a esteira que leva os caixotes até os recursos, que estão localizados em duas posições: R1 e R2. Sendo assim, essa classe conta com a skill `move`, que, de acordo com o valor passado como argumento, pode mover um caixote para a posição R1, R2 ou para o fim do módulo (isto é, próximo módulo);
- `MachineTool`: classe filha de `MRA` que representa a parte principal do módulo Conveyor Belt with Machine Tool, isto é, a parte que atua tampando os caixotes. Por isso, a skill dessa classe é a skill `cover`, que, caso exista um caixote posicionado, o tampa;
- `PneumaticPicking`: classe filha de `MRA` que modela o módulo Pneumatic Picking, que insere bolinhas nos caixotes. Tem somente a skill `insert`, que insere a quantidade de bolinhas de acordo com os valores passados como argumento;
- `DestinyConveyor`: classe filha de `MRA` responsável por modelar as esteiras de acesso ao armazém. Conta com a skill `receive`, que é responsável por ligar o motor da esteira de modo a receber um caixote de uma direção especificada através dos argumentos passados.

Para a camada cognitiva, tem-se:

- `Instantiator`: classe que descreve um agente instanciador. É um agente `MRA` da camada cognitiva responsável por instanciar outros agentes para os agentes

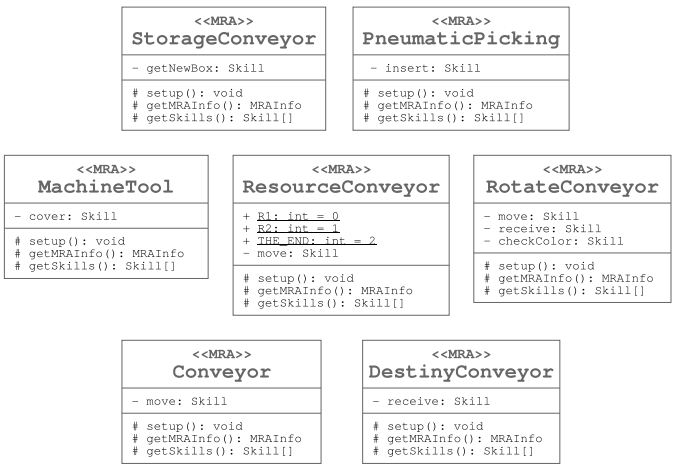

Figura 8. Classes da camada física.

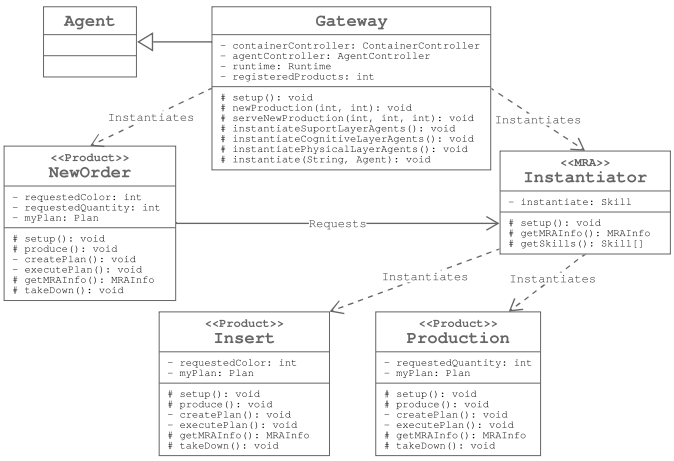

Figura 9. Classes da camada cognitiva e de negócios.

do tipo `Product` através da skill `instantiate`, a qual recebe uma `String` para identificar o nome do agente a ser instanciado, e outra para identificar de qual classe será tal agente;

- `NewOrder`: modela uma nova produção até que o caixote tenha sua cor identificada. Ao ser instanciada, recebe dois parâmetros: a quantidade de bolinhas e a cor do caixote. No seu plano de produção são chamadas as skills dos agentes da `StorageConveyor` e da primeira `RotateConveyor`, a fim de pegar um novo caixote, verificar sua cor e decidir se o caixote será utilizado numa nova produção ou inserção. A partir disso, o agente `NewOrder` solicita uma instanciação de um novo agente do tipo `Production` ou `Insert` ao agente `Instantiator`;
- `Insert`: é uma classe filha de `Product` que representa uma nova inserção. Ao ser instanciada, recebe a cor do caixote como parâmetro. Tal inserção é feita após um caixote de cor indesejada ser retirado e analisado, de forma que seja guardado no armazém para uma produção futura. Para isso, no plano de produção, o agente do tipo `Insert` faz as chamadas de skill necessárias para levar o caixote até a esteira de destino adequada;
- `Production`: é uma classe filha de `Product` que representa uma produção de fato. Ao ser instanciada, recebe a quantidade de bolinhas do produto como parâmetro. Tal produção é feita após um caixote da cor requisitada ser retirado e analisado, de forma que o mesmo seja imediatamente utilizado na produção. Para isso, no plano de produção, o agente do tipo `Production` faz as chamadas de skills necessárias para levar o caixote para receber bolinhas, ser tampado e chegar até à esteira de destino adequada.

Enquanto para a camada de suporte, tem-se:

- `YPA`: classe responsável por implementar um agente com o papel de páginas amarelas na arquitetura (Yellow Pages Agent). Tal classe possui métodos que implementam os serviços de busca por skill, registro e desregistro de agentes numa tabela de itens do tipo `MRAInfo`.

Por fim, na camada de negócios:

- `Gateway`: é a classe responsável por representar o agente Gateway da arquitetura. Conta com métodos que encapsulam toda a produção, nos quais são instanciados agentes cognitivos para os quais delegam-se distribuidamente as tarefas. Conta também com uma lista de pedidos na qual o método `addNewRequest()` adiciona um novo pedido de produção. Tal método recebe como parâmetro a cor do caixote e a quantidade de bolinhas desejadas. A execução da lista de pedidos é feita pelo método `executeRequests()`.

7. RESULTADO E CONCLUSÃO

O intuito da descrição do Staudinger é oferecer ao usuário final o serviço de produção de um ou mais produtos em série. O usuário, portanto, poderá, através de uma GUI (Interface Gráfica do Usuário), digitar a cor, o número de bolinhas e, em seguida, adicionar tal pedido na lista. Uma vez que tal lista esteja completa, o usuário poderá pedir para o sistema começar a produzir.

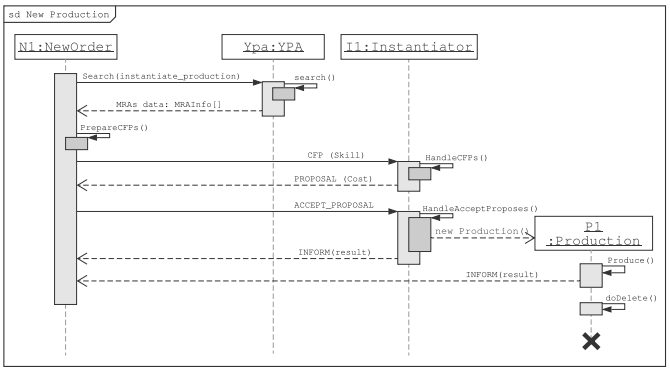

Figura 10. Sequência para uma nova produção.

Uma vez que for solicitado a produção, o agente `Gateway` irá instanciar um agente da classe `NewOrder`, para o qual uma tentativa de produção será delegada. Tal agente será responsável por levar o caixote da pilha até à esteira rotativa, onde será checado sua cor. Caso a cor coincida com a desejada para o pedido, o agente `NewOrder` solicitará a instanciação de um agente do tipo `Production` (Figura 10) para o `Instantiator`. Caso contrário, a solicitação de

instanciação será de um agente do tipo `Insert` (Figura 11). A Figura 12 ilustra tal processo.

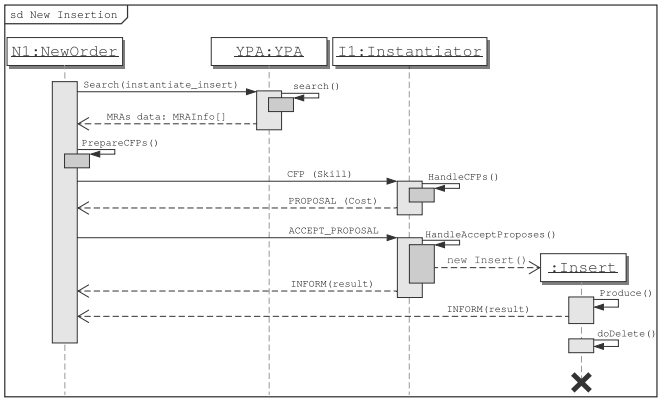

Figura 11. Sequência para uma nova inserção.

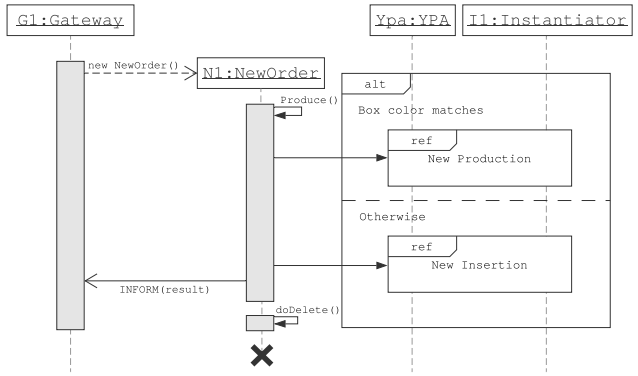

Figura 12. Sequência para uma nova tentativa produção.

Caso a tentativa de produção não dê certo, isto é, caso a cor do caixote não coincida, o agente `Gateway` irá tentar novamente repetindo o processo, ou seja, instanciando um novo agente `NewOrder`. Isso acontecerá indefinidamente até que a cor coincida e o pedido seja realizado ou o usuário cancele o pedido.

Portanto, o funcionamento adotado para o sistema foi alcançado com um sistema multiagente, que permite auto-organização. As habilidades (capacidades) de cada módulo físico do sistema poderiam ser oferecidas por outros módulos, que seriam inseridos para fins de análise de auto-organização, evidenciando as ditas (auto)formações de sociedades de agentes. Entretanto, o protótipo não permite retirada/inserção de novos módulos. Logo, para um trabalho futuro, tal análise de auto-organização deverá ser feita através de simulações.

REFERÊNCIAS

Alsterman, H. and Onori, M. (2004). Definitions, limitations and approaches of evolvable assembly system platforms. In *International Conference on Information Technology for Balanced Automation Systems*, 367–377. Springer.

Bellifemine, F., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with Jade*. Wiley.

Cavalcante, A.L.D. (2012). *Arquitetura Baseada em Agentes e Auto-organizável para a Manufatura*. Tese de Doutorado. UFRGS, Porto Alegre.

De Wolf, T. and Holvoet, T. (2004). Emergence and self-organisation: a statement of similarities and differences. In *Proceedings of the International Workshop on Engineering Self-Organising Applications 2004*, 96–110.

Frei, R., Ferreira, B., Serugendo, G.D.M., and Barata, J. (2009). An architecture for self-managing evolvable assembly systems. In *2009 IEEE International Conference on Systems, Man and Cybernetics*, 2707–2712. IEEE.

Koren, Y., Gu, X., and Guo, W. (2018). Reconfigurable manufacturing systems: Principles, design, and future trends. *Frontiers of Mechanical Engineering*, 13(2), 121–136.

Lee, E. (2007). Computing foundations and practice for cyber physical systems: A preliminary report.

Leitão, P., Karnouskos, S., Ribeiro, L., Lee, J., Strasser, T., and Colombo, A. (2016). Smart agents in industrial cyber–physical systems. *Proceedings of the IEEE*, 104, 1086–1101. doi:10.1109/JPROC.2016.2521931.

McArthur, S., Davidson, E., Catterson, V., Dimeas, A., Hatziargyriou, N., Ponci, F., and Funabashi, T. (2007). Multi-agent systems for power engineering applications—part i: Concepts, approaches, and technical challenges. *Power Systems, IEEE Transactions on*, 22, 1743 – 1752. doi:10.1109/TPWRS.2007.908471.

Mendonça, R.d.S., Cavalcante, A.L.D., and de Lucena Junior, V.F. (2016). Epscore: A didactic open architecture of an evolvable production system. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1–4. doi:10.1109/ETFA.2016.7733665.

Mendonça, R.d.S. (2016). *Plataforma Didática para Desenvolvimento de Sistemas Multiagente*. Dissertação de Mestrado. UFAM, Manaus.

Onori, M., Barata, J., and Frei, R. (2006). Evolvable assembly systems basic principles. In *International Conference on Information Technology for Balanced Automation Systems*, 317–328. Springer.

Sanderson, D., Turner, A., Shires, E., Chaplin, J., and Ratchev, S. (2019). Demonstration of transformable manufacturing systems through the evolvable assembly systems project. Technical report, SAE Technical Paper.

Suzić, N., Forza, C., Trentin, A., and Anišić, Z. (2018). Implementation guidelines for mass customization: current characteristics and suggestions for improvement. *Production Planning & Control*, 29(10), 856–871.