UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ALEX LUIZ DE SOUSA

CONTROLE DA PRODUÇÃO UTILIZANDO ESTRATÉGIAS DE CONSENSO MULTIAGENTE

TESE

CURITIBA

2023

ALEX LUIZ DE SOUSA

CONTROLE DA PRODUÇÃO UTILIZANDO ESTRATÉGIAS DE CONSENSO MULTIAGENTE

Production control using multi-agent consensus strategies

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, da Universidade Tecnológica Federal do Paraná (UTFPR), como requisito para a obtenção do título de Doutor em Ciências - Área de Concentração: Engenharia de Automação e Sistemas.

Orientador: Prof. Dr. André Schneider de Oliveira

CURITIBA

2023

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Curitiba

ALEX LUIZ DE SOUSA

CONTROLE DA PRODUÇÃO UTILIZANDO ESTRATÉGIAS DE CONSENSO MULTIAGENTE

Trabalho de pesquisa de doutorado apresentado como requisito para obtenção do título de Doutor Em Ciências da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Engenharia De Automação E Sistemas .

Data de aprovação: 24 de Outubro de 2023

Dr. Andre Schneider De Oliveira, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Cesar Augusto Tacla, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Douglas Wildgrube Bertol, Doutorado - Universidade do Estado de Santa Catarina - Udesc

Dr. Leandro Magatao, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Paulo Leitao, Doutorado - Instituto Politécnico de Bragança

Dedico este trabalho à minha esposa e filhos.

AGRADECIMENTOS

A Deus que me fortaleceu para chegar ao fim deste desafio.

Ao meu orientador Prof. André Schneider de Oliveira pelo apoio, sugestões e observações feitas no decorrer do doutorado.

À minha esposa Patrícia, e meus filhos Arthur e Matheus, pelo carinho e compreensão, que foram fundamentais para a conclusão deste trabalho.

Aos amigos Marco Antônio Simões Teixeira e Vivian Cremer Kalempa, pelo apoio e incentivo.

À UDESC pela oportunidade de realizar o doutorado com afastamento integral.

A todos os professores do departamento e colegas, que ajudaram de forma direta e indireta na conclusão deste trabalho.

Enfim, a todos os que de alguma forma contribuíram para a realização deste trabalho.

”Tudo posso naquele que me fortalece.”

– Filipenses 4:13 –

RESUMO

A flexibilidade nos processos produtivos, relacionada com a capacidade de reconfiguração das máquinas e a diversidade de produtos produzidos, é vista pela indústria como estratégia para se obter vantagem competitiva. Para garantir flexibilidade e reconfiguração, sistemas flexíveis de manufatura vem sendo atualizados por uma série de conceitos e tecnologias, representando um novo modelo de produção alinhado à Indústria 4.0. Entretanto, projetar controles inteligentes e adaptáveis para sistemas flexíveis de manufatura é um desafio. Abordagens empregando sistemas multiagentes têm sido bem-aceitas para este processo, devido à forma descentralizada e cooperativa com que os agentes resolvem problemas de otimização e lidam com tarefas complexas. O consenso multiagente, que pode ser visto como um procedimento algorítmico que possibilita convergência a novos estados de informação, pode ser explorado para compor soluções de controle mais eficientes em sistemas flexíveis de manufatura. Controlar a produção nestes sistemas requer que os agentes também possuam habilidades para lidar com situações imprevistas e indesejáveis, como ordens de urgência, *deadlocks* e outros impasses na produção. Muitas pesquisas propõe abordagens de reescalonamento como solução à urgências e impasses, mas se limitam a ambientes de produção específicos ou são difíceis de adaptação a outros cenários produtivos. Entretanto, certos tipos de impasses podem ser melhor resolvidos antecipadamente, no planejamento da produção, considerando o tipo e a capacidade do maquinário disponível. Restrições de *buffer* das máquinas e equipamentos, bem como o sistema de transporte utilizado para a movimentação de peças ou produtos entre as máquinas, precisam ser considerados no planejamento da produção. Ordens de urgência, que tem capacidade perturbadora significativa em tempo de produção, mas que são aceitas pela indústria devido ao mercado competitivo, também dependem de controles adequados às restrições do ambiente de produção. Além de impasses no sistema, a previsibilidade que é crucial para as indústrias atenderem as suas demandas de maneira eficiente, com custos menores e em níveis adequados, também corre o risco de ficar comprometida. Esta tese tem como finalidade gerar novas soluções para escalonamento e controle da produção utilizando estratégias de consenso multiagente e reajustamento de cronogramas preditivos. Os estudos desenvolvidos contemplam a flexibilidade nos processos produtivos, a descentralização da produção que é tendência da Indústria 4.0, soluções para problemas de impasses, condições de previsibilidade com escalonamento especializado, suporte à inserção de ordens urgentes em sistemas flexíveis de manufatura e a replicabilidade das abordagens para outros ambientes de produção. Experimentos comprovaram que sistemas multiagentes são eficientes para a descentralização de sistemas flexíveis de manufatura controlados por ordem. Filtros de reajustamento foram aplicados em cronogramas preditivos, a fim de evitar potenciais impasses na produção e as propostas de consenso multiagente garantiram a cooperação dos agentes em tarefas relacionadas às ordens de produção normais e urgentes. O aproveitamento do consenso para a inserção de ordens de urgência em tempo de produção, evitando impasses por sobreposição e por restrição de *buffer*, é uma inovação para sistemas flexíveis de manufatura e uma abordagem eficiente para evitar métodos computacionalmente onerosos de reescalonamento.

Palavras-chave: sistemas flexíveis de manufatura; sistemas multiagentes; escalonamento; prevenção de impasses; ordens de urgência.

ABSTRACT

Flexibility in production processes, related to the ability to reconfigure machines and the diversity of products produced, is seen by the industry as a strategy for gaining a competitive advantage. To ensure flexibility and reconfiguration, flexible manufacturing systems have been updated with concepts and technologies, representing a new production model aligned with Industry 4.0. However, designing intelligent and adaptable controls for flexible manufacturing systems is challenging. Approaches employing multi-agent systems have been well accepted for this process due to the decentralized and cooperative way agents solve optimization problems and deal with complex tasks. Multi-agent consensus, which can be seen as an algorithmic procedure that enables convergence to new information states, can be exploited to compose more efficient control solutions in flexible manufacturing systems. Controlling production in these systems requires that agents also have the skills to deal with unforeseen and undesirable situations, such as rush orders, deadlocks, and other production deadlocks. Much research proposes rescheduling approaches as a solution to rush orders and deadlocks, but they are limited to specific production environments or are difficult to adapt to other production scenarios. However, certain types of deadlock can be better resolved in advance in production planning, taking into account the type and capacity of machinery available. Machine and equipment buffer restrictions, as well as the transportation system used to move parts or products between machines, need to be taken into account when planning production. Rush orders, which have significant disruptive capacity in production time but are accepted by the industry due to the competitive market, also depend on adequate controls for the constraints of the production environment. In addition to deadlocks in the system, the predictability crucial for industries to meet their demands efficiently, at lower costs and at appropriate levels is also at risk of being compromised. The objective of this thesis is to generate new solutions for production scheduling and control using multi-agent consensus strategies and the readjustment of predictive schedules. The studies developed include flexibility in production processes, decentralization of production, a trend in Industry 4.0, solutions to deadlock problems, predictability conditions with specialized scheduling, support for the insertion of rush orders in flexible manufacturing systems, and the applicability of the approaches to other production environments. Experiments have proven that multi-agent systems efficiently decentralize order-controlled flexible manufacturing systems. Readjustment filters were applied to predictive schedules to avoid potential production deadlocks, and multi-agent consensus proposals ensured agent cooperation on tasks related to normal and rush production orders. Taking advantage of consensus to insert rush orders at production time, avoiding deadlocks due to overlapping and buffer constraints, is an innovation for flexible manufacturing systems and an efficient approach to avoiding computationally onerous rescheduling methods.

Keywords: flexible manufacturing systems; multi-agent systems; scheduling; deadlock-free; rush order.

LISTA DE ALGORITMOS

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

ABREVIATURAS

CBC	*Combiner control* – Controle combinador
CSC	*Classifier control* – Controle classificador
D-S	*Dempster-Shafer theory* – Teoria de Dempster-Shafer
DSC	*Color detection sensor* – Sensor de detecção de cor
OVF	*Overlay filter* – Filtro de sobreposição
PRF	*Pruning filter* – Filtro de podagem
SRF	*Serializer filter* – Filtro de serialização
S+P	*Serializer and pruning filters* – Filtros de serialização e de podagem

SIGLAS

FIPA	*Foundation for Intelligent Physical Agents* – Fundação para agentes físicos inteligentes
MQTT	*Message Queuing Telemetry Transport* – Transporte de telemetria de enfileiramento de mensagens
ROS	*Robot Operating System* – Sistema operacional de robôs

ACRÔNIMOS

CL	*Current location* – Localização atual
BTC	*Buffer and transportation constraints* – Restrições de transporte e de *buffer*
DSI	*Delivery station input* – Local de entrada de peças
DSO	*Delivery station output* – Local de saída de peças
FJS	*Flexible job-shop scheduling* – Escalonamento de *job-shop* flexível
FMC	*Flexible manufacturing cell* – Célula de manufatura flexível
FMS	*Flexible manufacturing system* – Sistema flexível de manufatura
HBW	*High-bay warehouse* – Armazém de alta capacidade
JS	*Job-shop scheduling* – Escalonamento de *job-shop*
MAS	*Multi-agent system* – Sistemas multiagentes
MPO	*Multi-processing station* – Estação de multiprocessamento
MS1–7	*Manufacturing stations* (1–7) – Estações de manufatura (de 1 até 7)
NFC	*Near-field communication* – Comunicação por proximidade de campo
NL	*Next location* – Próxima localização
OFTW	*Overlay free time window* – Janela de tempo livre de sobreposição
SLD	*Sorting line and detection* – Linha de classificação e detecção
VGR	*Vacuum gripper robot* – Robô com garra de sucção a vácuo

SUMÁRIO

1 INTRODUÇÃO

A transformação de materiais não processados em produtos acabados vem exigindo níveis cada vez maiores de automação e de flexibilidade na produção. A tecnologia atual possibilita que um mesmo equipamento seja aproveitado para fabricar produtos personalizados, ou de vários modelos, de acordo com as demandas dos clientes. Devido à competitividade de mercado, as indústrias estão investindo em ambientes de manufatura cada vez mais complexos, formados por múltiplas máquinas de produção, com o objetivo de oferecer maior capacidade de resposta e de personalização. A flexibilidade nos processos produtivos é vista pela indústria como uma estratégia para se obter vantagem competitiva. Para garantir flexibilidade e reconfiguração, os modernos sistemas de fabricação precisam evoluir suas arquiteturas em termos de inteligência (BOCCELLA *et al.*, 2020). Muitas pesquisas têm se concentrado no desenvolvimento de arquiteturas inteligentes para sistemas flexíveis de manufatura (em inglês, *Flexible Manufacturing Systems* - FMS) contando com conceitos e tecnologias alinhadas à Indústria 4.0. Os FMSs estão sendo atualizados e transformados por internet das coisas (IoT), *big data*, sistemas ciberfísicos, realidade virtual, computação em nuvem e várias outras tecnologias da Indústria 4.0, que surgiram para melhorar a flexibilidade em todo o sistema (JAVAID *et al.*, 2022).

Pequenas e médias empresas muitas vezes optam por investir em uma categoria de FMS chamada célula de manufatura flexível (em inglês, *Flexible Manufacturing Cell* - FMC). De acordo com Zhang *et al.* (2020b), um FMC introduz o conceito de "célul" como sendo parte de um FMS maior (apesar de autônomo), mas de menor custo financeiro e, por isso, é ideal para empresas que se encontram limitadas pelo capital. FMSs e FMCs podem fornecer uma rápida reação às mudanças inesperadas da produção porque contêm soluções tecnológicas eficientes para alcançar qualidade e flexibilidade (SELLITTO, 2020). No entanto, uma dificuldade no projeto de controles para FMSs é a complexidade desses sistemas devido ao comportamento estocástico caracterizado por tempos de processamento aleatórios, variabilidade da produção, alto número de eventos simultâneos, planejamento de produção sob incertezas, *deadlocks* e atrasos. É importante que os FMSs sejam adaptados com a ajuda de infraestruturas inteligentes, flexíveis e dinâmicas, para se reconfigurarem rapidamente e reagirem em casos de mudanças previstas ou imprevistas (por exemplo, ordens de produção urgentes ou impasses).

Os desafios da introdução de abordagens de controle de fabricação adaptáveis e flexíveis podem ser resolvidos com o uso de sistemas distribuídos apoiados por unidades autônomas que

cooperam entre si (BARENJI *et al.*, 2014). Estratégias distribuídas para controle de sistemas de manufatura complexos foram propostas para aumentar a flexibilidade nos processos de fabricação, como em Calà *et al.* (2016), Bi *et al.* (2021), Diaz e Ocampo-Martinez (2021). Uma estratégia de controle distribuído que pode ser explorada para coordenar sistemas complexos é o controle multiagente (WOOLDRIDGE, 2009). Sistemas multiagentes (do inglês, *Multi-agent Systems* - MAS) são excepcionalmente adequados para modelar problemas complexos devido à possibilidade de dividir o problema em partes menores e encapsular funcionalidades. O MAS representa uma abordagem promissora para o desenvolvimento de controles para FMSs devido à forma descentralizada e cooperativa com que os agentes resolvem problemas de otimização e lidam com tarefas complexas (DENKENA *et al.*, 2021). Trabalhar cooperativamente requer que os agentes se comuniquem e cheguem a um "consenso" (SOUSA; OLIVEIRA, 2020).

O consenso pode ser visto como um procedimento algorítmico que permite a convergência de estado entre agentes localmente autônomos que colaboram para um objetivo comum (MEZGEBE *et al.*, 2020). Segundo Weiss (2013), a negociação, argumentação, votação, leilões e coalizões são métodos utilizados para alcançar consenso em cenários cooperativos e competitivos. O consenso é a base do controle coordenado distribuído de um MAS e tem sido amplamente utilizado em controle cooperativo (LI; TAN, 2019). No entanto, a aplicabilidade do consenso para problemas de controle na manufatura, especificamente para problemas de escalonamento da produção, ainda não foi exaustivamente estudada. De acordo com Mezgebe *et al.* (2020), os algoritmos de consenso raramente foram adaptados a algoritmos de tomada de decisão ou ainda não foram implementados em FMSs. Com um controle de consenso adequado para FMSs com produção controlada por ordem, os agentes podem cooperar entre si seguindo cronogramas de produção com especificações de logística e de fabricação organizadas por um processo de escalonamento. O escalonamento da produção é fundamental para o sucesso de um FMS.

Um dos problemas de escalonamento da produção mais explorados é o *job-shop scheduling* (JS). JS é um problema de otimização combinatória que visa atribuir um conjunto de trabalhos (*jobs*) a um conjunto de máquinas, geralmente para minimizar o tempo máximo de conclusão de todos os trabalhos (ZHANG *et al.*, 2020a; HAM, 2020). Em um problema de JS, uma máquina pode processar no máximo uma operação por vez. Cada trabalho, por sua vez, é composto de um conjunto de operações sequenciais. Um tempo de processamento é definido para cada operação e, uma vez iniciada a operação, ela não pode ser interrompida (não é preemptiva) (MOGALI *et al.*, 2021; XIE *et al.*, 2019). Entretanto, problemas de escalonamento em FMSs

são melhor tratados por uma variante do JS conhecida como *flexible job-shop scheduling* (FJS). Em um FJS, assume-se que uma operação pode ser processada por qualquer máquina de um determinado conjunto de máquinas disponíveis (LI *et al.*, 2020; MOGALI *et al.*, 2021; LI *et al.*, 2021); a recirculação de trabalhos também é permitida (GAO *et al.*, 2020). Os trabalhos podem ser atribuídos a algumas máquinas mais de uma vez (máquinas polivalentes) de acordo com a capacidade de flexibilidade do sistema. Sistemas de manufatura automatizados, especialmente FMSs, podem ser considerados uma generalização de problemas de FJS com *buffers*[1] de capacidade limitada (GAO *et al.*, 2020).

A elaboração de um planejamento da produção eficaz muitas vezes envolve um grande número de entidades e interações integradas que, se mal planejadas, podem levar o FMS à impasses severos. Impasses podem ser temporários ou por tempo indefinido (*deadlocks*) e ambos resultam em prejuízos para as indústrias. Um impasse também pode significar um período de tempo em que um produto permanece na máquina de processamento, resultando na incapacidade da máquina de realizar qualquer outro trabalho (LUO *et al.*, 2020; SUN *et al.*, 2021). Além disso, vários produtos podem entrar em produção em pontos separados no tempo para processamento em uma sequência específica de máquinas e operações. Ordens de urgência são comuns em sistemas de manufatura e são prioritárias às ordens normais em execução, mas tem sua inserção condicionada à disponibilidade dos recursos de produção (e capacidade dos *buffers*). Portanto, lidar com o roteamento de produtos e a capacidade de *buffer* do sistema é fundamental para evitar impasses. De acordo com Han *et al.* (2020), os impasses restringem severamente o processo de produção e aumentam a dificuldade de controle.

1.1 CONTEXTUALIZAÇÃO E MOTIVAÇÃO

O escalonamento da produção é um tópico clássico no controle de manufatura e é crucial para melhorar a utilização de recursos e a eficiência do FMS (UHLMANN; FRAZZON, 2018; HU *et al.*, 2020a). Para resolver de forma eficiente problemas complexos de alocação de recursos e de sequenciamento de tarefas, as soluções de escalonamento da produção devem estar alinhadas com a capacidade reativa do FMS. Obter as melhores soluções para os problemas de escalonamento é de grande importância para a indústria, uma vez que a taxa de produção e as despesas no ambiente fabril dependem dos cronogramas utilizados. Um cronograma de produção assertivo permite que a indústria empregue seus recursos de maneira eficaz e atinja

[1] *Buffers* são locais na linha de produção onde peças podem ser temporariamente armazenadas.

os objetivos estratégicos esperados. Portanto, é relevante para as indústrias que problemas específicos de escalonamento possam ser resolvidos diretamente nos cronogramas de produção, através de otimizações, priorizações e implementação de ações que melhorem a produtividade, sem interferir no controle do FMS. Essas são questões cruciais de produção que podem ser resolvidas previamente, no planejamento da produção.

Embora muitos problemas de escalonamento tenham sido amplamente pesquisados, grande parte das pesquisas consideram espaços de *buffer* infinitos no sistema para reduzir a complexidade da solução (GAO *et al.*, 2020; SUN *et al.*, 2021). No entanto, na maioria dos cenários do mundo real, *buffer* ilimitado não é uma expectativa realista já que podem haver limites físicos na capacidade dos recursos (MOGALI *et al.*, 2021). Consequentemente, problemas de escalonamento com *buffers* de capacidade limitada são pouco estudados, ficando ignoradas questões como sobrecarga de recursos e impasses no sistema. Algoritmos de escalonamento de *job-shops* geralmente não são capazes de identificar e prevenir potenciais impasses durante a geração dos cronogramas de produção. Entretanto, é adequado que restrições de *buffer* e de transporte sejam tratadas no escalonamento, considerando o tipo de maquinário disponível no chão de fábrica e o roteamento de insumos entre eles. Há equipamentos, por exemplo, que não dispõe de *buffer*, ou que os próprios equipamentos servem de *buffer* temporário até que outros equipamentos estejam disponíveis. Em alguns casos o sistema de transporte também pode ter outra função agregada além de movimentar peças entre equipamentos (por exemplo, posicionar a peça em um equipamento antes de transferir para outro).

A habilidade para lidar com ordens de urgência evitando impasses na produção também é crucial em FMSs. Uma ordem de urgência é um evento inesperado bastante comum, mas que exige capacidade de resposta do FMS e, por vários motivos, deve ser atendida mais rapidamente, inclusive sobrepondo-se ao fluxo de produção normal do sistema. Geralmente, demandas sob ordens de urgência estão relacionadas com o mercado competitivo e a estratégia da indústria para maximizar sua economia, além de ampliar e manter a fidelidade dos clientes. As literaturas têm considerado a ordem de urgência como um distúrbio especial que interrompe o desempenho de todo o chão de fábrica (WANG *et al.*, 2022). Para Mezgebe *et al.* (2020), uma ordem de urgência é um evento que tem capacidade perturbadora significativa para um cronograma preditivo centralizado. Abordagens inteligentes para suporte à urgência e impasses por limitações de *buffer* e de transporte são cruciais para FMSs com produção controlada por ordem. Na literatura é bem-aceito que soluções baseadas em MAS são eficientes para garantir flexibilidade e tratar

impasses na produção (SOUSA; OLIVEIRA, 2022). De acordo com Mezgebe *et al.* (2020), as habilidades dos agentes geralmente são utilizadas no escalonamento da produção para melhorar a qualidade das soluções finais.

MAS já provaram sua eficiência apresentando características de auto-organização para trabalhos cooperativos, como em (SOUSA; OLIVEIRA, 2020). Devido a sua flexibilidade em relação à comunicação, MAS também é uma escolha apropriada para problemas dinâmicos onde a configuração do sistema pode mudar em tempo de execução (GEHLHOFF; FAY, 2020). Por exemplo, eventos inesperados podem ocorrer no FMS a qualquer momento, exigindo ações de controle e decisões mais ágeis e eficientes. A implantação de um MAS pode favorecer a escalabilidade e a tolerância à falhas no FMS, pois a carga de processamento pode ser dividida entre agentes. A capacidade de escalabilidade permite que o sistema evolua conforme o necessário, quando estendida para máquinas adicionais para aumento da capacidade de processamento na produção, ou como redundância à equipamentos críticos. Do contrário, se a carga de processamento é concentrada em um único ponto de controle centralizado, o controle pode não conseguir tratar de forma satisfatória todos os eventos recebidos conforme o número de elementos no FMS aumenta (por exemplo, com a conexão de um novo FMC na planta). A tolerância à falhas em um MAS distribuído também é maior, pois o gerenciamento do sistema não depende de um único controlador central que pode falhar ou ficar indisponível. Além disso, um MAS operando de forma distribuída tem sua capacidade de processamento ampliada, possibilitando maior rapidez em tomadas de decisões e na convergência do FMS para novas soluções de controle. Um MAS distribuído é dinâmico no sentido de que agentes podem juntar-se ou deixar de executar operações no FMS a qualquer momento, sem impactos na performance ou topologia; por exemplo, quando um agente associado a uma determinada máquina é alocado para uma operação de produção.

A literatura apresenta estudos com soluções importantes para problemas de escalonamento, impasses e outros tipos de interrupções na produção. Entretanto, algumas soluções se limitam a resultados teóricos baseados em modelos simplificados de FMS, tornando-se inviáveis para aplicação em ambientes reais ou em outros cenários de pesquisa. Além disso, as simplificações acabam negligenciando processos importantes para o escalonamento, como a capacidade de *buffer* ou o roteamento de produtos no sistema. Desta forma, a lacuna identificada na literatura está em apresentar abordagens que proporcionem flexibilidade nos processos produtivos, garantindo uma produção livres de impasses, aplicável em diferentes cenários produtivos e realista com as necessidades da indústria.

A seguir, o objetivo geral da tese e os objetivos específicos são apresentados. Cada etapa dos trabalhos realizados durante o doutorado está relacionada com um objetivo específico.

1.2 OBJETIVOS

- O objetivo geral da tese é apresentar uma pesquisa exploratória sobre sistemas multiagentes, sistemas flexíveis de manufatura e escalonamento da produção, bem como propostas para flexibilidade na produção, escalonamento livre de impasses e inserção de ordens de urgência, motivadas pelos estudos desenvolvidos na temática de escalonamento e controle da produção empregando consenso multiagente.

1.2.1 Objetivos específicos

1. *Pesquisa para as contribuições da tese*. Realizar pesquisa exploratória sobre sistemas multiagentes, sistemas flexíveis de manufatura e escalonamento da produção. Determinar os pontos em aberto das técnicas, as abordagens promissoras e justificar os pontos de inovação das contribuições propostas pela tese.

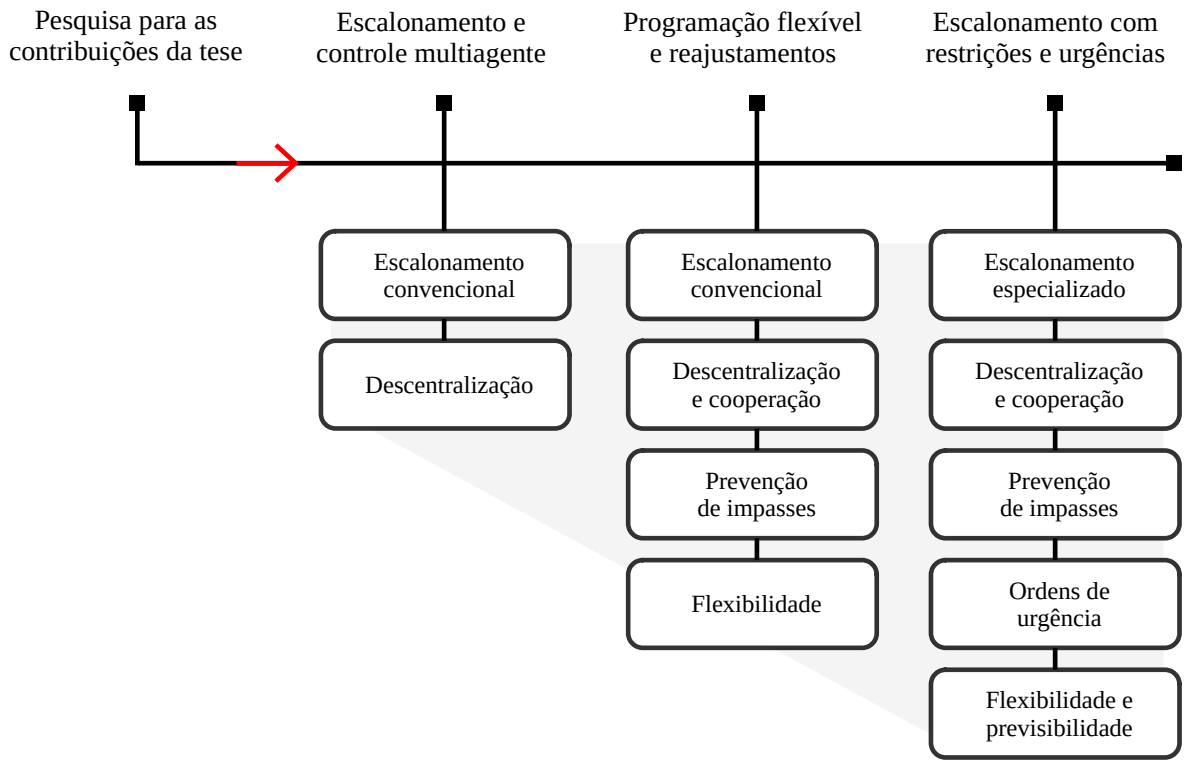
2. *Escalonamento e controle multiagente*. Propor um MAS distribuído e abordagens de controle da produção para experimentação. Os agentes devem encaminhar os produtos para estações de manufatura específicas, conforme cronogramas de produção gerados por um algoritmo de *job-shop* flexível, e aplicar métodos de controle para evitar certos tipos de impasses. O consenso deve garantir a cooperação dos agentes nas atividades de produção.

3. *Programação flexível e reajustamentos*. Propor uma arquitetura MAS capaz de suportar cronogramas preditivos reajustados, para garantir uma produção flexível e livre de impasse. Os reajustes devem ocorrer no planejamento da produção, em conjunto com o escalonamento, e de forma transparente para o MAS. O sistema de manufatura deve ser flexível o suficiente para suportar cronogramas de produção com múltiplos produtos.

4. *Escalonamento com restrições e urgências*. Propor uma nova abordagem de escalonamento para problemas com restrições por limitação de *buffer* e de transporte. A abordagem deve ser combinada a consenso multiagente e suportar a inserção de ordens de urgência em *runtime*. O MAS deve ser capaz de operar com cronogramas preditivos livres de impasses, incluindo os possíveis impasses ocasionados pelas ordens de urgência.

A Figura 1 apresenta o relacionamento entre os estudos desenvolvidos na tese. A partir das pesquisas para as contribuições da tese, iniciou-se um estudo preliminar explorando o emprego de sistemas multiagentes para a descentralização do controle da produção. O escalonamento é suplementar, pois atua na geração de cronogramas para que o controle execute a produção seguindo corretamente as especificações de fabricação dos itens produzidos. O estudo seguinte trabalha melhor o controle, aprimorando a cooperação entre agentes via consenso. Além disso, o estudo se concentrou no reajustamento dos cronogramas para prevenir certos impasses e garantir flexibilidade no sistema produtivo. O último estudo deu maior ênfase ao consenso, aproveitando os períodos de convergência dos agentes para a inclusão de novas funcionalidade no sistema (suporte à ordens urgentes). Por fim, um processo escalonamento especializado foi aplicado na criação dos cronogramas para fornecer maior previsibilidade na produção. O escalonamento conduzido nos estudos é preditivo (centralizado), ou seja, é baseado em uma visão completa e inicial do estado do sistema. Portanto, os cronogramas são previamente gerados e enviados para o MAS, que aplica controle descentralizado na produção.

Figura 1 – Relacionamento entre os estudos desenvolvidos na tese.



Fonte: Autoria própria.

As estratégias utilizando consenso multiagente se limitam ao controle da produção,

objetivando a descentralização, com flexibilidade nos processos produtivos em termos de escolha das máquinas de produção de acordo com o produto produzido. Neste sentido, considera-se a produção multimáquina e multiproduto, ou seja, várias máquinas trabalhando juntas para produzir um produto final, possibilitando a fabricação de diferentes produtos simultaneamente. Com efeito, a decisão consensual possibilitará garantir a completa execução de cronogramas livres de impasses e de ordens de produção prioritárias no sistema.

1.3 TRABALHOS PUBLICADOS E EM REVISÃO

1. SOUSA, Alex L.; OLIVEIRA, André S. *Distributed MAS with Leaderless Consensus to Job-shop Scheduler in a Virtual Smart Factory with Modular Conveyors*. Submetido para: 17th IEEE Latin American Robotics Symposium (LARS). 2020, p. 1–6. Status: publicado.

2. SOUSA, Alex L.; OLIVEIRA, André S. *Deadlock-free Production Scheduling in a MAS-controlled FMS with Dempster-Shafer Classifier and Preset Combiner methods*. Submetido para: Applied Soft Computing. 2022. p. 1–32. Status: em revisão.

3. SOUSA, Alex L.; OLIVEIRA, André S. *Order-Controlled Production Employing Multi-Agent and Flexible Job-Shop Scheduling on a Physical Simulation Platform*. Submetido para: 19th IEEE Latin American Robotics Symposium (LARS). 2022. p. 1–6. Status: publicado.

4. SOUSA, Alex L.; OLIVEIRA, André S. *Finite-Time Consensus and Readjustment Three-Stage Filter for Predictive Schedules in FMS*. Submetido para: IEEE Access. 2023. vol 11, p. 88558–88582. Status: publicado.

5. SOUSA, Alex L.; OLIVEIRA, André S. *Proposta de Consenso Multiagente para Validação de Cronogramas Preditivos de Urgência em Sistemas Flexíveis de Manufatura*. Submetido para: XVI Simpósio Brasileiro de Automação Inteligente (SBAI 2023). 2023. p. 1–6. Status: publicado.

6. SOUSA, Alex L.; OLIVEIRA, André S. *Proposal for a Scheduler with Buffer and Transportation Constraints to Insert Rush Orders without Deadlocks in Agent Convergence*. Submetido para: Computers & Industrial Engineering. 2023. p. 1–22. Status: em revisão.

1.4 ORGANIZAÇÃO DO DOCUMENTO

Este documento está dividido em seis capítulos. O Capítulo 1 introduz os temas abordados nos estudos desenvolvidos na tese e apresenta o objetivo geral e objetivos específicos. O Capítulo 2 apresenta o estado da arte, com uma revisão da literatura sobre sistemas multiagentes, sistemas flexíveis de manufatura e escalonamento da produção. O Capítulo 3 apresenta um estudo envolvendo multiagentes na descentralização do controle da produção. O Capítulo 4 apresenta um estudo sobre programação flexível da produção e reajustamento de cronogramas para a prevenção de impasses. O Capítulo 5 trata do escalonamento com restrições e urgências, e o aproveitamento do consenso multiagente no controle. Por fim, o Capítulo 6 apresenta as conclusões finais do documento de tese.

2 ESTADO DA ARTE

Este capítulo apresenta uma pesquisa exploratória sobre temas fundamentais para os estudos desenvolvidos e documentados na tese. O primeiro tema trata de sistemas flexíveis de manufatura e apresenta características, classificação e tipos de problemas relacionados com a implantação destes sistemas. O segundo tema apresenta conceitos sobre sistemas multiagentes, metodologias de projeto e as principais linhas de pesquisa. Na sequência, são discutidos problemas de escalonamento e impasses na produção, principais abordagens e pontos em aberto das técnicas empregadas. Por fim, são apresentadas as conclusões do capítulo.

2.1 SISTEMAS FLEXÍVEIS DE MANUFATURA

Em decorrência dos avanços tecnológicos e demandas de mercado, máquinas com múltiplas funções estão cada vez mais presentes nas indústrias. A capacidade de produção de itens diversificados a partir de um único equipamento possibilitou o surgimento de ambientes fabris com múltiplas máquinas de produção e paralelismo de produtos. Desde então, muitos pesquisadores vem concentrando esforços no aperfeiçoamento de sistemas flexíveis de manufatura para o gerenciamento da produção. Sistemas flexíveis de manufatura se caracterizam como uma forma de produção com altos níveis de automatização, capazes de se adaptarem e prontamente reagirem alterando os processos produtivos conforme necessário.

2.1.1 Conceitos sobre FMS

Os sistemas flexíveis de manufatura (em inglês, *Flexible Manufacturing System* - FMS) surgiram baseados na ideia de “flexibilidade”, e ganharam reconhecimento devido ao seu grande potencial de adaptabilidade. Para Groover (2020), um FMS é um sistema de manufatura altamente automatizado, que consiste em múltiplas estações de processamento, interconectadas por um sistema automatizado de manuseio e armazenamento de materiais e controladas por um sistema de computador integrado. Atualmente, FMSs representam um novo modelo de produção alinhado ao contexto da Indústria 4.0 para enfrentar os desafios do mercado contemporâneo. De acordo com Zhang *et al.* (2020b) e Far *et al.* (2018), os FMSs podem ser classificados em categorias, conforme suas formas estruturais: *Single Flexible Machine* - SFM, *Flexible Manu-*

facturing Cell - FMC, *Multi-Machine Flexible Manufacturing System* - MMFMS e *Multi-Cell Flexible Manufacturing System* - MCFMS. Esta classificação deixa claro que SFMs e FMCs são as principais unidades de produção, enquanto que MMFMSs e MCFMSs são combinações das anteriores. Um SFM é definido como uma "unidade de produção formada por máquinas de controle numérico (NC), completada pela facilidade de manipulação para alterar os objetos da produção" (HAJDUK *et al.*, 2018). De acordo com Lee *et al.* (2020), as máquinas possuem trocadores automáticos de ferramentas e, portanto, operações consecutivas podem ser realizadas com tempos de configuração insignificantes. Já os FMCs, Hajduk *et al.* (2018) explicam que consistem em agrupamentos de várias máquinas NC, específicas para a fabricação de um determinado grupo de produtos com sequências de operação semelhantes, ou para um determinado tipo de operação; o sinal característico da célula é a interconexão mútua de material e de informação entre as máquinas.

Atualizar um sistema de manufatura tradicional para flexível não é uma tarefa simples. A implantação de FMSs envolvem, em geral, três problemas principais que devem ser bem estudados e definidos para se obter a flexibilidade e eficiência desejadas:

- *Problemas de planejamento*. Nos problemas de planejamento, geralmente agrupam-se máquinas semelhantes nas mesmas células de máquina, e produtos ou peças com roteamento e processamento semelhantes nas mesmas famílias de produtos. De acordo com Forghani e Fatemi Ghomi (2020), este processo de decisão estratégica da fábrica é denominado formação de célula (em inglês, *cell formation* - CF), e consiste em encontrar o *layout* das máquinas dentro das células (*layout* intracelular) e o *layout* das células no chão de fábrica (*layout* intercelular). Um planejamento de *layout* adequado possibilita aumentar a eficiência e a flexibilidade em qualquer ambiente de fabricação (KHAMLICHI *et al.*, 2020). De acordo com Sadeghi *et al.* (2020), estes agrupamentos de células são conduzidos a partir de um conceito conhecido como tecnologia de grupo (em inglês, *group technology* - GT). Buscar minimizar os tempos de roteamento de peças inter e intracelular, durante a formação das células, é fundamental para o escalonamento.

- *Problemas de escalonamento*. Concluídos os processos de CF e GT, o escalonamento de células (em inglês, *cell scheduling* - CS) é a próxima estratégia a ser tomada, pois envolve a alocação dos recursos de produção na fábrica. O problema surge quando recursos limitados do sistema devem ser otimizados para atender aos objetivos de produção (KHAMLICHI *et al.*, 2020). CS é uma decisão tática sobre como processar os produtos nas máquinas durante

cada período de produção (FORGHANI; Fatemi Ghomi, 2020). As células podem estar organizadas em paralelo ou de outras formas, equipadas com vários tipos de máquinas heterogêneas, e processar diferentes tipos de produtos de forma simultânea e independente (WU *et al.*, 2021). Definir sequências de operações de *jobs* que maximizem a produção e minimizem atrasos, evitando potenciais impasses nas operações de fabricação, são critérios de desempenho essenciais para os FMSs.

- *Problemas de controle*. Problemas de controle estão relacionados com o monitoramento do sistema e o rastreamento da produção, de forma que os requisitos de produção sejam atendidos de acordo com o escalonamento (LEE *et al.*, 2020). Tornar o sistema mais eficiente e flexível, evitando que desvios das condições normais de operação ocorram, é fundamental para o controle de FMSs. De acordo com Boccella *et al.* (2020), FMSs devem ser capazes de lidar com falhas ou quebras de máquinas e capazes de verificar seu próprio estado, bem como a disponibilidade de seus recursos. A flexibilidade em um FMS é percebida pela sua capacidade em alterar facilmente configurações para se ajustar às necessidades de produção (MOURTZIS *et al.*, 2020). A garantia de flexibilidade é maior quando os recursos de roteamento e a capacidade de *buffer* do sistema estão incluídos nas soluções de escalonamento.

Para Hansmann e Hoeck (1997), FMSs podem ser vistos como uma forma altamente automatizada para instalações de produção multiuso, como as instalações de *job-shops*. De acordo com o WordSense Dictionary[1], *job-shops* são instalações especializadas na produção ou fabricação de peças em quantidades relativamente pequenas, produzidas de acordo com as especificações ou exigências fornecidas pelos clientes. A formulação clássica de problemas de escalonamento de *job-shop* é baseada na suposição de que para cada tipo de peça (*job*) ou ordem de produção existe apenas um cronograma que prescreve a sequência de máquinas e operações (HANSMANN; HOECK, 1997). Entretanto, com os modernos equipamentos de fabricação e sistemas de controle inteligentes, cronogramas de produção alternativos podem estar disponíveis, já que a flexibilidade dos atuais ambientes de produção é maior.

[1] https://www.wordsense.eu/job_shop/

2.1.2 Problemas de *job-shop* flexível

O *job-shop scheduling* (JS) está entre os problemas mais bem estudados de escalonamento (ou programação) da produção (LI *et al.*, 2022) e é considerado um problema de otimização combinatória NP-difícil (LIU *et al.*, 2020). Em termos de complexidade computacional, JS é NP-difícil no sentido forte e, mesmo para instâncias JS muito pequenas, uma solução ótima não pode ser garantida (CHAUDHRY; KHAN, 2016). *Flexible job-shop scheduling* (FJS) é uma extensão do problema JS clássico e dos ambientes de máquinas paralelas (PINEDO, 2022). Diferente do JS, o FJS permite que uma operação seja processada por qualquer máquina de um determinado conjunto de máquinas alternativas. Portanto, de acordo com Xie *et al.* (2019), Chaudhry e Khan (2016), além de lidar com o sequenciamento das operações o FJS também precisa atribuir máquinas específicas para as operações (o roteamento das operações dos *jobs*). No FJS, um conjunto de n jobs $J = \{J_1, J_2, ..., J_n\}$ deve ser processado em um conjunto de m máquinas $M = \{M_1, M_2, ..., M_m\}$. Cada job J_i possui uma sequência de j operações $O_{ij} = \{O_{i1}, O_{i2}, ..., O_{ij}\}$, que são específicas do processo de fabricação de cada produto. De acordo com Li *et al.* (2020), cada operação deve selecionar uma máquina disponível de um conjunto de máquinas candidatas, de modo que ela seja processada em apenas uma máquina por vez. A máquina não pode ser ocupada novamente até que a operação atribuída a ela seja concluída, ou seja, não há preempção.

A maioria das pesquisas com *job-shop scheduling* se concentram em escalonamento centralizado ou escalonamento semi-distribuído (ZHANG *et al.*, 2019). A pesquisa de Abdolrazzagh-Nezhad e Abdullah (2017) faz uma classificação e apresenta restrições e função objetivo impostas aos problemas de JS disponíveis na literatura. Em geral, o objetivo do escalonamento é definir sequências de operações de *jobs* (peças ou produtos) em cada máquina, para otimização de certas variáveis sob condições de restrição. Contudo, a descentralização da produção é uma tendência da Indústria 4.0 para os modernos sistemas de manufatura. Além disso, as soluções de escalonamento devem estar alinhadas a métodos de controle inteligentes para garantir eficiência e flexibilidade ao FMS. Os desafios de introduzir abordagens de controle inteligentes, descentralizadas, adaptáveis e flexíveis podem ser explorados com sistemas multiagentes.

2.2 SISTEMAS MULTIAGENTES

Uma ampla gama de tarefas pode ser resolvida com maior eficácia em trabalhos cooperativos, onde cada entidade autônoma pode desempenhar o papel de um "agente" encarregado de realizar parte do trabalho. Para Figat e Zieliński (2020), um agente pode ser visto como um sistema que afeta racionalmente o seu entorno se baseando em informações coletadas do meio ambiente. Russell e Norvig (2016) definiram um agente como "uma entidade autônoma flexível capaz de perceber o ambiente através de sensores conectados a ele e atuar nesse ambiente através de atuadores". A literatura denomina de "sistema multiagente" (do inglês, *Multi-agent System* - MAS) quando há o envolvimento de múltiplos agentes nas soluções de pesquisa. Sistemas multiagentes têm se destacado como uma abordagem promissora para desenvolver controles inteligentes e cooperativos, despertando o interesse de muitos pesquisadores nos últimos anos (MALASCHUK; DYMIN, 2020; AFANASYEV *et al.*, 2019; ISMAIL; SARIFF, 2019).

2.2.1 Conceitos sobre MAS

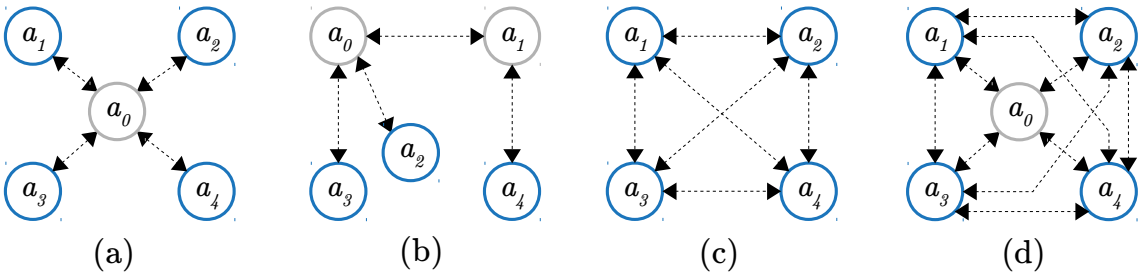
Em um MAS os agentes podem ser divididos em duas categorias, denominadas homogênea e heterogênea (EFREMOV; KHOLOD, 2020; ROSS *et al.*, 2019; DORRI *et al.*, 2018). Os *agentes homogêneos* possuem suas estruturas ou capacidades físicas idênticas, ao contrário de *agentes heterogêneos* que não são idênticas e cada agente pode ter sua própria especialização ou tarefa específica para concluir. Algumas pesquisas provaram que agentes homogêneos podem executar tarefas com eficiência, e que a cobertura das tarefas para agentes homogêneos é máxima em comparação com agentes heterogêneos (ISMAIL; SARIFF, 2019; SINGH *et al.*, 2018). Da mesma forma, pesquisas provaram que agentes heterogêneos são mais aplicáveis do que homogêneos em muitas situações no mundo real (ZHANG; ZHU, 2018; LUO; LIU, 2019). No estudo de Borisov *et al.* (2018) os pesquisadores empregaram agentes heterogêneos no desenvolvimento de um sistema ciberfísico industrial. Na pesquisa de Miah *et al.* (2018), bons resultados foram alcançados com agentes homogêneos em um sistema de proteção para ambientes estruturados.

Independente dos tipos de agentes envolvidos no MAS, a coordenação do sistema geralmente é baseada em três abordagens de controle (SINGHAL *et al.*, 2018; JIMéNEZ *et al.*, 2018): centralizada, descentralizada e distribuída. O *controle centralizado* conta com elevado nível de comunicação, sensoriamento e representação completa do ambiente (ISMAIL; SARIFF, 2019). Porém, segundo Soumya e Guruprasad (2020), essa abordagem sofre de alta carga

computacional e sobrecarga de comunicação. O controlador fica sujeito a falhas que podem comprometer todo o sistema. Na abordagem de *controle descentralizado* o agente processa suas informações localmente (JIMéNEZ *et al.*, 2018). Cada agente é orientado por um sistema de controle individual e não há interação entre os controladores, embora os agentes possam interagir entre si (SOUMYA; GURUPRASAD, 2020). Singhal *et al.* (2018) argumentam que na abordagem descentralizada podem haver vários nós de processamento, tornando o sistema mais robusto e escalável. No *controle distribuído*, os agentes podem cooperar entre si com base na percepção direta, transmissão de sinais, ou comunicação indireta por meio de mudanças ambientais (ISMAIL; SARIFF, 2019). Para Singhal *et al.* (2018), a abordagem distribuída é usada em tarefas que requerem maior conhecimento do ambiente e autonomia para tomadas de decisão pelos próprios agentes.

A integração entre o controle centralizado e o controle distribuído é conhecida como *controle híbrido* (VOROTNIKOV *et al.*, 2018; SEREBRENNY *et al.*, 2019; VIKSNIN *et al.*, 2019). De acordo com Ismail e Sariff (2019), geralmente a parte de controle centralizado está envolvida nos cálculos de uma meta global (um exemplo, otimizar o *makespan*), enquanto que a parte de controle distribuído é mais voltada ao plano local (por exemplo, a montagem de uma peça). A solução de problemas complexos em condições de incerteza requer principalmente sistemas de controle do tipo híbrido (VOROTNIKOV *et al.*, 2018). A Figura 2 exibe as diferentes abordagens de controle para MAS.

Figura 2 – Diferentes abordagens para controle de MAS.



Fonte: Autoria própria.

Na abordagem centralizada (Figura 2a), cada agente se comunica com o coordenador centralizado (a_0), que na prática é um agente ou outro sistema de controle. O coordenador centralizado toma e comunica as decisões a todos os agentes sob sua coordenação. Na abordagem de controle descentralizada, há mais de um coordenador que toma e comunica decisões (a_0 e a_1) a grupos específicos de agentes (Figura 2b); cada agente se comunica com seu respectivo

coordenador. Uma abordagem distribuída com quatro agentes é apresentada na Figura 2c, onde cada agente toma suas próprias decisões com base na comunicação com outros agentes e informações do ambiente. Em uma abordagem de controle híbrida, todos os agentes geralmente se comunicam entre si (Figura 2d), de modo que a comunicação impacta no comportamento do agente. No entanto, o coordenador centralizado (a_0) é quem envia o plano final para cada agente.

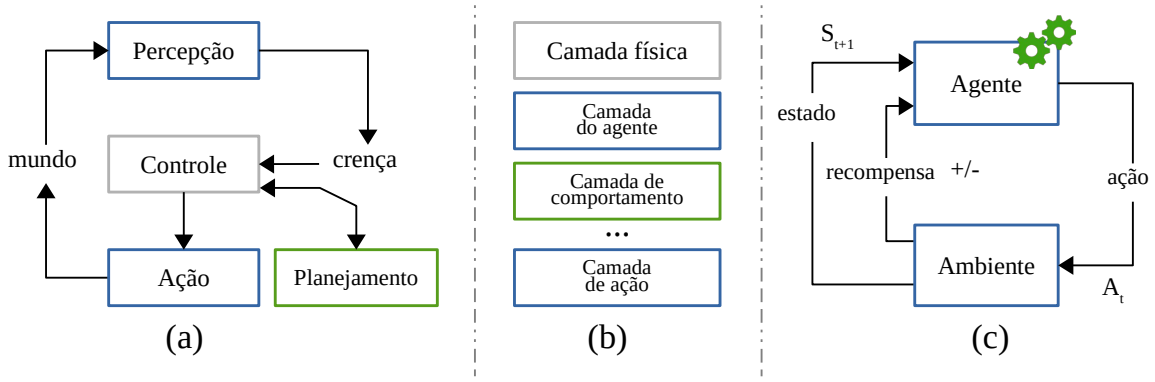
Os agentes são capazes de executar tarefas através de comunicações ativas entre si, porém fornecer comunicações confiáveis em um MAS distribuído é um trabalho complexo (VIKSNIN *et al.*, 2019). Há dois tipos de comunicação importantes para MAS, a comunicação implícita e a comunicação explícita. A *comunicação implícita* se refere à capacidade dos agentes detectarem uns aos outros incorporando tipos diferentes de sensores entre eles (ISMAIL; SARIFF, 2019). Já a *comunicação explícita*, refere-se à troca direta de informações entre agentes via algum módulo de comunicação embarcado (ISMAIL; SARIFF, 2019). Entretanto, nestas duas formas de comunicação surgem problemas relacionados com limitações de hardware e desafios no *design* das topologias de rede e do protocolo de comunicação (por exemplo, problemas de segurança). Pesquisas com foco em segurança da informação visando a transferência de dados críticos entre agentes podem ser vistas em (SHUMSKAYA; ISKHAKOVA, 2019; VIKSNIN *et al.*, 2019). Uma proposta de estrutura de comunicação que não necessita um vínculo de comunicação permanente entre agentes é apresentada em (JIMéNEZ *et al.*, 2018).

2.2.2 Metodologias de projeto

Há muitas visões diferentes sobre como projetar sistemas envolvendo múltiplos agentes devido ao fato de ser um campo altamente interdisciplinar (WOOLDRIDGE, 2009). Alguns pesquisadores seguem a *abordagem clássica* da Robótica (Figura 3a), realizando a decomposição primária da percepção, planejamento e ação dos agentes, como em (QUINTERO *et al.*, 2012). Por outro lado, outras abordagens defendem o projeto de um sistema de controle *camada por camada* (Figura 3b), como em (FIGAT; ZIELIŃSKI, 2020) e (LIEKNA *et al.*, 2013). Uma metodologia alternativa é utilizar explicitamente *técnicas evolutivas* para evoluir gradualmente sistemas complexos de controle (MIYAWAKI *et al.*, 2018). Por exemplo, *Q-learning* é uma abordagem que geralmente possui um conjunto de estados e ações para cada estado. Ao mudar de estado (S_{t+1}), o agente recebe uma recompensa positiva ou negativa (Figura 3c); o objetivo do agente é maximizar sua recompensa total.

Quanto às abordagens para descrever e modelar sistemas multiagentes, as metodologias

Figura 3 – Diferentes metodologias para projeto de MAS.



Fonte: Autoria própria.

da Engenharia de Software (itens 1-3, Tabela 1) são bem-aceitas porque estão mais consolidadas e difundidas, especialmente os diagramas UML. No entanto, modelos matemáticos para descrever o comportamento dos agentes são mais proeminentes, como mostram os itens 4-6 da Tabela 1. Em geral, estas representações e teorias matemáticas já englobam as abordagens MAS utilizadas nos problemas de pesquisa. O uso de modelos próprios de representação também são bem-aceitos (item 7), pois facilitam o entendimento para pesquisadores de diferentes áreas do conhecimento.

Tabela 1 – Abordagens para descrever e modelar MAS.

Item	Abordagens / Referências	Qtd.
1	Diagramas UML (DARINTSEV *et al.*, 2019; GARCíA *et al.*, 2012; STOLLEIS, 2015; VIKSNIN *et al.*, 2019; BRAHMI *et al.*, 2019)	5
2	Diagramas de bloco de função (BRAHMI *et al.*, 2019; ZUO *et al.*, 2019)	2
3	Fluxogramas (ABIDIN *et al.*, 2012; ABBASI *et al.*, 2018; SOUSA; OLIVEIRA, 2020; SRIVASTAVA *et al.*, 2013; NAJJAR *et al.*, 2019)	5
4	Teoria dos grafos (DENG; YANG, 2019; LUO; LIU, 2019; OH *et al.*, 2015; PABON; MOJICA-NAVA, 2019; WANG; WANG, 2017)	5
5	Redes de Petri (FIGAT; ZIELIŃSKI, 2020; ROSA *et al.*, 2015; GARCíA *et al.*, 2012)	3
6	Outros modelos matemáticos (AHMED *et al.*, 2014; AHSAN; MA, 2019; OMAR; PAYEUR, 2019; MIAH *et al.*, 2018; NORRAZI *et al.*, 2013; DENG; YANG, 2019; FIGAT; ZIELIŃSKI, 2020; OH *et al.*, 2015; RAHIMI *et al.*, 2014; REZAEE; ABDOLLAHI, 2019; ROSS *et al.*, 2019; WANG; WANG, 2017; YANG *et al.*, 2018; YAO *et al.*, 2018; ZHANG; ZHU, 2018; GANZHUR *et al.*, 2019; GARCíA *et al.*, 2012; PEREVERZEVA *et al.*, 2012; PIERPAOLI *et al.*, 2018; PULJIZ *et al.*, 2012; TING *et al.*, 2012)	21
7	Modelos próprios de representação (BOCCELLA *et al.*, 2020; BORISOV *et al.*, 2018; BOUTERAA; DERBEL, 2018; CHEUNG *et al.*, 2011; LIEKNA *et al.*, 2013; NORRAZI *et al.*, 2013; PIARDI *et al.*, 2019; SOUSA; OLIVEIRA, 2020; NAJJAR *et al.*, 2019; ABIDIN *et al.*, 2012; GIL *et al.*, 2019; JIMéNEZ *et al.*, 2018; DUCHOň *et al.*, 2016)	13

Fonte: Autoria própria.

Na literatura, há diversas abordagens que tentam desenvolver ou estender as metodologias tradicionais de engenharia de software e do conhecimento, para a especificação e projeto

de MAS. Em 2002, a *Foundation for Intelligent Physical Agents* (FIPA) concluiu um processo de padronização envolvendo um conjunto de especificações para MAS. Essas especificações representam um conjunto de padrões que visam a interoperação de agentes heterogêneos e os serviços que eles podem representar (FIPA, 2004). Depois que os agentes e demais requisitos do MAS são especificados, inicia-se a etapa de implementação e experimentação do sistema com algum tipo de plataforma. Geralmente a plataforma já é considerada na fase de projeto, como prevê as especificações FIPA. As principais funções de uma plataforma de agentes são garantir comunicação confiável e possibilitar o rápido desenvolvimento do sistema. Um exemplo de plataforma para multiagentes utilizada por algumas pesquisas é JADE (*Java Agent Development framework*). Dorri *et al.* (2018) mencionam outra plataforma chamada GAMA, que possibilita a modelagem/simulação de MAS, suportando GAML (uma linguagem para descrever o comportamento de agentes) e MAS em larga escala. A plataforma GAMA é mais atual que JADE, que teve sua última *release* em 2017.

2.2.3 Principais linhas de pesquisa

Na literatura de 2010 a 2023, as pesquisas selecionadas que envolvem MAS apresentam uma série de abordagens para os mais variados objetivos. Com base nos objetivos dos artigos analisados, as principais linhas de pesquisa para MAS foram levantadas (Tabela 2). Estudos relacionados com mais de uma linha de pesquisa também foram contabilizados. Por exemplo, o trabalho de Afanasyev *et al.* (2019) propôs um método de alocação de tarefas e um método de controle de consenso para MAS utilizando *blockchains*.

As principais linhas de pesquisa em MAS são *controle de consenso*, *controle de contenção*, *controle de formação*, *alocação de tarefas*, *estrutura de comunicação* e *métodos de aprendizagem*. Os itens enumerados a seguir apresentam as respectivas linhas de pesquisa e destacam algumas referências relacionadas.

1. *Controle de consenso*. O desenvolvimento de protocolos de consenso tem sido estudado intensivamente por pesquisadores. De acordo com Ismail e Sariff (2019), o controle de consenso pode ser classificado em *líder-seguidor* e *consenso sem líderes*, tanto para agentes homogêneos quanto para heterogêneos. Porém, encontrar consenso entre agentes heterogêneos é mais desafiador, já que os agentes e seus estados são diferentes entre si. Cheung *et al.* (2011) apresentam uma proposta de controle de consenso utilizando um método

Tabela 2 – Principais linhas de pesquisa em MAS.

Item	Linhas de pesquisa / Referências	Qtd.
1	Controle de consenso (AFANASYEV *et al.*, 2019; GIL *et al.*, 2019; DENG; YANG, 2019; KALEMPA *et al.*, 2020; LUO; LIU, 2019; OH *et al.*, 2015; PARK *et al.*, 2013; PIERPAOLI *et al.*, 2018; SOUSA; OLIVEIRA, 2020; ZHANG; ZHU, 2018; SHUTIN; ZHANG, 2016)	11
2	Controle de contenção (AHSAN; MA, 2019; SHAO *et al.*, 2015; WANG; WANG, 2017; YANG *et al.*, 2018)	4
3	Controle de formação (ABIDIN *et al.*, 2012; AHMED *et al.*, 2014; CHEUNG *et al.*, 2011; FRANCESCHELLI *et al.*, 2010; KARIGIANNIS *et al.*, 2011; MIAH *et al.*, 2018; NORRAZI *et al.*, 2013; OH *et al.*, 2015; PIERPAOLI *et al.*, 2018; RAHIMI *et al.*, 2014; REZAEE; ABDOLLAHI, 2019; ROSA *et al.*, 2015; SRIVASTAVA *et al.*, 2013)	13
4	Alocação de tarefas (AFANASYEV *et al.*, 2019; OMAR; PAYEUR, 2019; DARINTSEV *et al.*, 2019; KALEMPA *et al.*, 2021; LAVENDELIS *et al.*, 2012; LIEKNA *et al.*, 2013; ROSA *et al.*, 2015; ROSS *et al.*, 2019; SINGHAL *et al.*, 2018; SOUSA; OLIVEIRA, 2020; HE; SHI, 2015)	11
5	Estrutura de comunicação (CHOUHAN; NIYOGI, 2012; DUCHOň *et al.*, 2016; FIGAT; ZIELIŃSKI, 2020; GANZHUR *et al.*, 2019; GARCíA *et al.*, 2012; JIMéNEZ *et al.*, 2018; KAPITONOV *et al.*, 2019; MALASCHUK; DYMIN, 2020; MAVROGIANNIS; KNEPPER, 2021; NAJJAR *et al.*, 2019; SRIVASTAVA *et al.*, 2013; VIKSNIN *et al.*, 2019; WEST, 2017)	13
6	Métodos de aprendizagem (ABBASI *et al.*, 2018; GIL *et al.*, 2019; EFREMOV; KHOLOD, 2020; KARIGIANNIS *et al.*, 2011; LUO; LIU, 2019; SHAW *et al.*, 2022; SHUTIN; ZHANG, 2016; STOLLEIS, 2015; TING *et al.*, 2012; WEST, 2017; YANG *et al.*, 2018)	11

Fonte: Autoria própria.

baseado em campos potenciais e um algoritmo de leilão para consenso líder-seguidor; em Rosa *et al.* (2015), também desenvolveram um método de consenso líder-seguidor para a atribuição de tarefas. Em Pabon e Mojica-Nava (2019), os pesquisadores propõe uma abordagem para consenso líder-seguidor impulsionada por eventos. Na pesquisa de Najjar *et al.* (2019), é proposto um protocolo de comunicação para consenso líder-seguidor utilizando hardware de baixo custo. Já na pesquisa de Rezaee e Abdollahi (2019), empregaram topologia cíclica (sem líder) com uma abordagem de campo vetorial para o consenso. Sistemas Lagrangianos foram utilizados para sincronizar os estados em que o consenso é alcançado (BOUTERAA; DERBEL, 2018).

2. *Controle de contenção*. O controle de contenção refere-se à introdução de mais de um líder entre os agentes, para garantir que os grupos de agentes não corram riscos (ISMAIL; SARIFF, 2019), por exemplo, em um ambiente perigoso. Se os agentes enfrentarem essa situação, eles serão guiados para uma região segura, chamada de fecho convexo ou envoltória convexa (em inglês, *convex hull*), delimitada por um grupo de líderes (SHAO *et al.*, 2015). De acordo com Yang *et al.* (2018), os agentes líderes têm maior capacidade de percepção de situações de risco ou indesejadas e podem guiar outros agentes seguidores no cumprimento da missão. O trabalho de Zhang e Zhu (2018) utiliza a teoria da estabilidade

de Lyapunov para lidar com o problema de contenção. Uma proposta de controladores de contenção distribuídos com base na teoria de Hamilton pode ser vista no trabalho de Shao *et al.* (2015). A pesquisa de Wang e Wang (2017) propõe um controle de contenção por tempo finito com *líderes estáticos* e *dinâmicos*. Controle de contenção bipartido para sistemas multiagentes de primeira e de segunda ordem[2] é discutido em Ahsan e Ma (2019).

3. *Controle de formação*. O controle de formação está relacionado com a capacidade dos agentes em controlar suas posições e orientação relativas entre os demais membros de um grupo (por exemplo, robôs), a fim de moverem-se para um ponto específico (ISMAIL; SARIFF, 2019; REZAEE; ABDOLLAHI, 2019). Na literatura existem três principais estratégias de controle de formação (RAHIMI *et al.*, 2014; ISMAIL; SARIFF, 2019; NORRAZI *et al.*, 2013): estratégia *líder-seguidor*, onde um agente é escolhido como líder para seguir uma trajetória, enquanto que os demais devem manter distância do líder e realizar uma formação predefinida, seguindo-o; estratégia com *estrutura virtual*, que é semelhante à estratégia de líder-seguidor, no entanto o líder é virtual e nunca falha, favorecendo a estabilidade do sistema; estratégia *baseada em comportamento*, que se refere à prescrição de vários comportamentos desejados aos agentes, tal como coesão, prevenção de colisões, prevenção de obstáculos, dentre outros (OH *et al.*, 2015).

4. *Alocação de tarefas*. Depois que o sistema atribui uma tarefa, ela precisa ser enviada aos agentes para execução. Segundo Ross *et al.* (2019), determinar quais agentes são adequados ou capazes de resolver uma determinada tarefa pode ser desafiador e é conhecido na literatura como problema de alocação de tarefas. O sistema de controle é responsável por decompor as tarefas e atribuir (ou reatribuir em caso de falhas) para os agentes mais adequados (LAVENDELIS *et al.*, 2012). Tarefas maiores geralmente são divididas em subtarefas menores, que podem ser trabalhadas por agentes individuais ou equipes de cooperação (ROSS *et al.*, 2019). A pesquisa de Ismail e Sariff (2019) utiliza conceitos de *Attractive Field Model* (AFM) e otimização multiobjetivo (ou otimização de Pareto) para auto-organizar agentes e alocar tarefas. Algoritmos de otimização combinatória, processos de decisão markovianos, bem como técnicas estocásticas ou probabilísticas de atribuição de tarefas foram estudadas por Omar e Payeur (2019). Uma proposta de arquitetura de

[2] A ordem (primeira, segunda, ou superior) refere-se ao número de métricas que os agentes devem concordar para chegar a um consenso (DORRI *et al.*, 2018).

controle para a alocação de tarefas, empregando conceitos de redes de Petri na definição de tarefas cooperativas complexas, foi apresentada em Rosa *et al.* (2015).

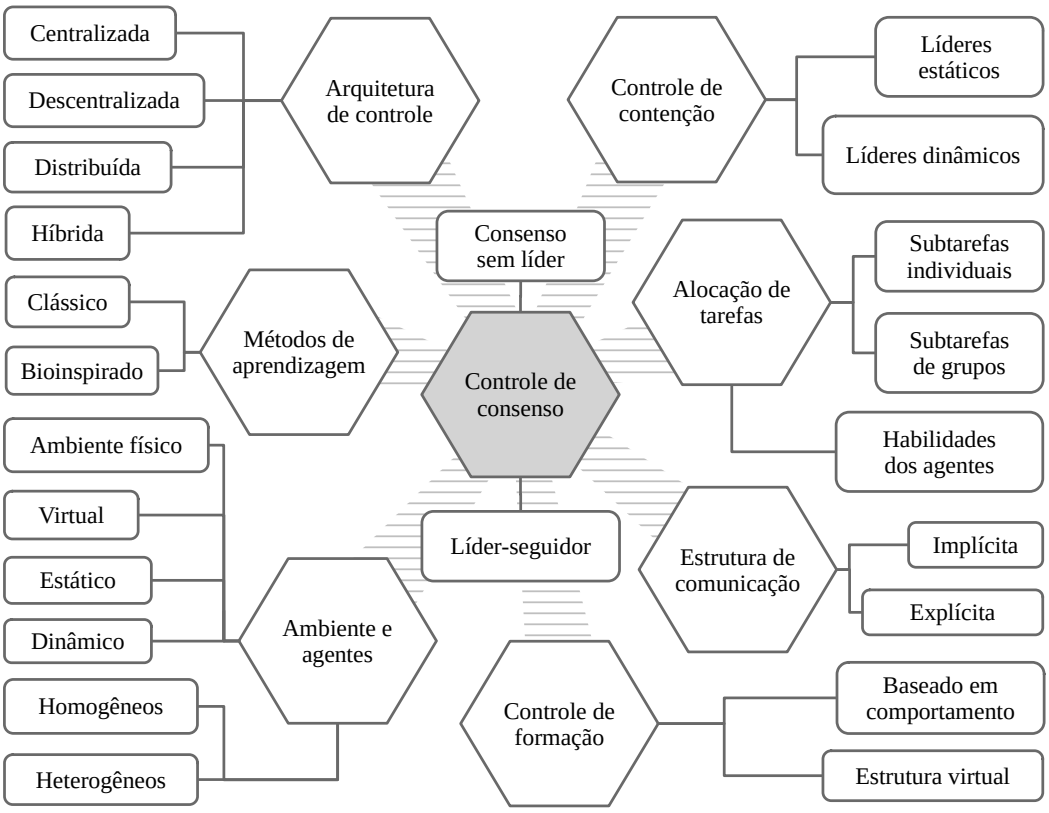
5. *Estrutura de comunicação*. Prover meios de comunicação adequados para múltiplos agentes é essencial para o sucesso do MAS. Ainda que a *comunicação explícita* seja mais empregada, alguns pesquisadores buscam coordenar múltiplos agentes provendo-os da capacidade de reconhecer comportamentos. Por exemplo, Mavrogiannis e Knepper (2021) exploraram a navegação descentralizada entre múltiplos agentes não comunicantes utilizando mecanismos de previsão de comportamentos baseados em princípios de Hamilton, inferência simbólica e otimização de gradiente. De acordo com Ismail e Sariff (2019), alguns pesquisadores combinam *comunicações implícitas* e explícitas em suas pesquisas. Um exemplo é a pesquisa de Liekna *et al.* (2013), que trabalha no desenvolvimento de sistemas multiagentes de tempo real, baseados em comportamento. Com relação à segurança e sobrecarga de comunicação, algumas pesquisas apresentaram soluções importantes para multiagentes, cita-se (JIMéNEZ *et al.*, 2018; VIKSNIN *et al.*, 2019; SHUMSKAYA; ISKHAKOVA, 2019). Seguindo esta mesma linha, Ganzhur *et al.* (2019) propuseram um modelo teórico de segurança baseado em níveis de confiança e reputação entre agentes, para proteger o MAS contra intrusões. Outro ponto crítico para o sucesso do MAS é a definição da topologia de rede. A pesquisa de Srivastava *et al.* (2013) apresenta uma proposta de sistema de localização baseado na indicação da intensidade do sinal recebido (RSSI), onde os agentes se comunicam usando redes *mash*. No entanto, no trabalho de Jiménez *et al.* (2018), os pesquisadores dizem que a abordagem ideal para comunicação é através de uma rede de sensores sem fios já que os agentes geralmente são compostos por unidades de processamento independentes, módulos de comunicação sem fios e sensores.

6. *Métodos de aprendizagem*. Para melhorar a aptidão de um agente em um ambiente dinâmico, o aprendizado, a evolução e a adaptação são importantes. A "inteligência" dos agentes para trabalharem cooperativamente ou coordenarem suas tarefas é baseada em seu sistema de controle; o projeto do controlador determinará o desempenho do agente (ISMAIL; SARIFF, 2019). Vários métodos de aprendizagem para MAS foram pesquisados, Karigiannis *et al.* (2011) apresentam uma proposta de aprendizagem por reforço (ou programação neurodinâmica). No trabalho de Abidin *et al.* (2012) a aprendizagem dos agentes é inspirada na "mosca da fruta", objetivando resolver problemas de mapeamento. Algoritmos genéticos para exploração espacial (TING *et al.*, 2012), aprendizado bayesiano

esparso para um problema de consenso e retorno à base (SHUTIN; ZHANG, 2016), IA distribuída para problemas de colisão (SORIANO *et al.*, 2013) e métodos básicos de inteligência artificial (IA) para alocação de tarefas (DARINTSEV *et al.*, 2019), também foram aplicados para a obtenção de comportamentos cooperativos.

Encontrar valores precisos para os parâmetros de controle que levam ao comportamento cooperativo desejado em um MAS pode ser uma tarefa demorada e desafiadora. Considerando que os agentes são sistemas situados em algum ambiente físico ou virtual, estático ou dinâmico, é possível determinar certas propriedades que eles possuem. Em quase todas as aplicações práticas os agentes têm, na melhor das hipóteses, controle parcial sobre seu ambiente. Assim, embora possam realizar ações que mudem seu ambiente, geralmente não podem controlá-lo totalmente, mas podem cooperar para obter controle total. Dito isso, considera-se que o controle de consenso é o ponto central de um MAS e a base para outras linhas de pesquisa (Figura 4).

Figura 4 – Resumo da pesquisa em MAS, com destaque para consenso.



Fonte: Autoria própria.

É fundamental desenvolver mecanismos de consenso ao lidar com controle de contenção, controle de formação ou alocação de tarefas, devido à dependência de consenso nestas linhas de pesquisa. Os estudos envolvendo métodos de aprendizagem geralmente estão associados

com abordagens que também dependem da obtenção de consenso, mas a aprendizagem é mais enfatizada nos estudos. Da mesma forma, pesquisas relacionadas com estruturas de comunicação buscam melhorar a coordenação do MAS e, de forma indireta, o consenso. Por exemplo, capacidade de resposta, tolerância à falhas, segurança ou protocolos de controle. Em todos os casos, o consenso é determinante para que os agentes desenvolvam tarefas cooperativas e habilidades para reagirem a eventos.

O aproveitamento do consenso na convergência dos agentes para novos estados de informação também possibilita o desenvolvimento de soluções mais sofisticadas e robustas para problemas de produção. Quando a convergência ocorre dentro de um intervalo de tempo finito, onde o tempo exato de obtenção do consenso é parâmetro *a priori* do protocolo de consenso e os agentes com quaisquer valores iniciais arbitrários concordam em um estado consistente, é chamado de "consenso por tempo finito pré-definido" (AMIRKHANI; BARSHOOI, 2022). De acordo com Li e Tan (2019), muitos sistemas de controle práticos requerem tempo de convergência mais rigoroso e resposta dinâmica rápida, que são vantagens do consenso multiagente por tempo finito.

Dentro da noção de consenso completo, onde todos os agentes convergem para um valor comum, há a noção de consenso de grupo (ou consenso de *cluster*). Neste tipo de consenso os agentes são divididos em diferentes grupos, e cada grupo convergirá para um valor de consenso diferente (AMIRKHANI; BARSHOOI, 2022). Em muitas situações práticas, um grupo de agentes deve ser capaz de sentir e responder à situações inesperadas ou quaisquer alterações quando uma tarefa cooperativa é implementada (LI; TAN, 2019). De acordo com Amirkhani e Barshooi (2022), cada grupo pode ser considerado como um sistema separado e forma um consenso completo, cujo ponto de equilíbrio não precisa se adaptar ao dos outros grupos. O consenso de grupo é adequado para lidar com problemas de controle cooperativo (LI; TAN, 2019). Para Ji *et al.* (2021), todo o sistema pode alcançar consenso por meio dos grupos, o que é especialmente adequado para controle coordenado em MAS de grande escala.

2.2.4 Benefícios do MAS para sistemas de manufatura

Na manufatura há uma demanda crescente por produtos cada vez mais customizados. Entretanto, para atender o mercado contemporâneo, é necessário flexibilizar o fluxo de produção integrando todas as partes envolvidas nos processos de manufatura. Para gerenciar mudanças no fluxo produtivo e tornar a manufatura auto-organizável, MAS tem se apresentado como a melhor

solução para este desafio. Há diversos campos em que MAS são aplicados com sucesso e para sistemas de manufatura complexos eles também podem trazer uma série de benefícios. Alguns desses benefícios incluem:

- A decomposição de grandes problemas em partes menores e, potencialmente, mais facilmente gerenciáveis pelos agentes, que podem ser programados para encapsular certas funcionalidades para lidar com o problema.
- Descentralização e modularidade, possibilitando que os processos de produção sejam executados de acordo com a demanda. Maior flexibilidade para alterar operações e, além de receber comandos, as máquinas podem fornecer informações operacionais aos agentes.
- A complexidade toma forma hierárquica, ou seja, um sistema é composto de sub-sistemas inter-relacionados que, por sua vez, também apresentam uma hierarquia. Logo, a descentralização com MAS possibilita que níveis hierárquicos mais baixos tomem decisões.
- Capacidade de reação e capacidade pró-ativa, com agentes capazes de perceber e reagir rapidamente à alterações no ambiente de produção em que estiverem inseridos, bem como tomar iniciativa quando julgarem apropriado aos seus objetivos.

Estas características diferem o MAS de outros sistemas distribuídos usuais, ao mesmo tempo em que também proporcionam escalabilidade para lidar com tarefas maiores e complexas, maior robustez e capacidade de tolerar falhas de agentes individuais. Um MAS pode ajudar a aumentar a eficiência de um FMS automatizando tarefas, melhorando a precisão do sistema, aumentando a flexibilidade e a adaptação à mudanças. Os agentes também podem ser projetados para trabalhar com informações e conhecimentos incertos ou incompletos.

O uso de MAS como meio de interação entre os componentes de um sistema de manufatura se apresenta como uma solução à necessidade de auto-organização, proporcionando maior flexibilidade e autonomia no processo produtivo. Com a adesão de novas tecnologias da Indústria 4.0, a dependência de controles centralizados é menor, reduzindo riscos de falhas e proporcionando maior disponibilidade do sistema. Os agentes possibilitam que os recursos sejam descentralizados para dar origem a produtos e serviços customizados, de forma que a indústria possa atender às novas demandas fabris.

2.3 ESCALONAMENTO E IMPASSES NA PRODUÇÃO

A combinação de características de FMSs e *job-shops* possibilita o surgimento de importantes ambientes de produção capazes de produzir uma ampla variedade de produtos. Com o apoio de um MAS distribuído, as operações de produção podem ser distribuídas entre vários agentes proporcionando maior flexibilidade ao FMS. De acordo com Ismail e Sariff (2019), multiagentes tornam o controle distribuído mais escalável, adaptável, flexível e robusto. Na pesquisa de Fazlirad e Brennan (2018), os principais esquemas de escalonamento multiagentes sugeridos na literatura são apresentados. Segundo os pesquisadores, novas abordagens e técnicas combinadas com MAS têm se mostrado promissoras no escalonamento. Entretanto, garantir cronogramas livres de impasses em ambientes de fabricação dinâmicos e com paralelismo de produtos é uma tarefa desafiadora.

A pesquisa de Meilanitasari e Shin (2021) revisou outros métodos modernos de previsão e otimização no escalonamento de *job-shops* para FMSs. Porém, os pesquisadores identificaram que a busca por maior agilidade e flexibilidade para alguns FMSs recém-desenvolvidos resulta em um número significativo de incertezas que precisam ser quantificadas e modeladas explicitamente. O escalonamento sob incertezas é propício a impasses que geralmente afetam as métricas de desempenho desejadas e, por isso, o FMS deve ser capaz de identificá-los e evitá-los. São exemplos de impasses alguns atrasos e a completa paralisação da produção (*deadlock*), ambos limitam o processo produtivo e a controlabilidade do sistema. Prover meios eficazes de controle de impasses para FMSs é crucial para garantir a otimização dos processos de produção e evitar prejuízos para as indústrias.

De acordo com a literatura revisada, vários estudos apresentam soluções significativas para determinadas situações de impasse. Alguns pesquisadores adotam modelos de redes de Petri (do inglês, *Petri nets* - PN) como formalismo para descrever FMSs e desenvolver políticas de prevenção de impasses, com em (HU *et al.*, 2020a; LUO *et al.*, 2020; MESSINIS; VOSNIAKOS, 2020; HU *et al.*, 2020b; NABI; AIZED, 2020; DU *et al.*, 2020; KAID *et al.*, 2020; BASHIR, 2020; ČAPKOVIč, 2023). Muitas dessas técnicas são baseadas na detecção de sifões (bloqueios estruturais) como os apresentados em (KAID *et al.*, 2020; BASHIR, 2020). Na pesquisa de Kaid *et al.* (2020), foi proposta uma abordagem de controle de *deadlocks* baseada em sifões mínimos estritos (em inglês, *strict minimal siphons* - SMS). Os locais de controle obtidos com os SMSs foram mesclados em um único local de controle usando redes de Petri coloridas. A pesquisa

de Bashir (2020) assemelha-se à pesquisa de Kaid. No entanto, o método proposto por Bashir foi utilizado para calcular zonas de trabalho livres de *deadlocks* e controles para uma estrutura supervisória de FMS descentralizada. Já no trabalho de Hu *et al.* (2020a), os pesquisadores combinaram técnicas de aprendizagem por reforço (em inglês, *deep reinforcement learning* - DRL) com redes de Petri temporizadas, para reduzir o número de parâmetros de treinamento e evitar *deadlocks* em uma solução de escalonamento dinâmico para um FMS.

Pesquisas envolvendo escalonamento dinâmico (por exemplo, reescalonamento), que incluem métodos para recuperação de certos impasses, podem ser encontradas em (BAER *et al.*, 2019; ZHANG *et al.*, 2020a; LI *et al.*, 2021; HAN *et al.*, 2020; DURASEVIć; JAKOBOVIć, 2020; LUO, 2020; KIANPOUR *et al.*, 2021; LIANG *et al.*, 2020; CALDEIRA *et al.*, 2020; XU; CHEN, 2022). Em geral, essas abordagens empregam técnicas heurísticas e IA, como algoritmos evolutivos ou redes neurais, treinadas para identificar o plano de reescalonamento mais viável e gerar novos cronogramas de produção.

Outro método amplamente aplicado em sistemas dinâmicos e flexíveis são as regras de despacho (em inglês, *dispatching rules* - DRs), consideradas como "decisões de escalonamento" para resolver problemas de produção. A pesquisa de Teymourifar *et al.* (2020) propôs uma abordagem para extrair regras eficientes para um tipo de FJS, onde os jobs chegam em momentos diferentes no sistema. Quebras de máquinas que ocorrem estocasticamente e condições de *buffer* limitado foram consideradas na pesquisa. Os pesquisadores combinaram a programação de expressão gênica (em inglês, *gene expression programming* - GEP) com um modelo de simulação para projetar DRs nas políticas de escalonamento. O estudo de Durasević e Jakobović (2020) foi semelhante ao de Teymourifar, os pesquisadores propuseram um método dividido em duas partes para criar DRs dinâmicas. Na pesquisa de Luo (2020) um método usando DQNs, treinado com uma *deep q-learning* aprimorada (um tipo de DRL), foi empregado para determinar a DR mais adequada de um conjunto de DRs disponíveis para problemas de escalonamento.

Alguns métodos para resolver impasses procuram trocar tarefas entre máquinas (e *buffers*). Na literatura existem duas versões de problemas de JS com restrição de bloqueio: bloqueio sem troca, e com troca. Nos casos de bloqueio sem troca, a restrição de capacidade do *buffer* (ou *buffer* infinito) é substituída por uma restrição de capacidade zero. Pesquisas sobre problemas de *job-shop* com restrições de bloqueio (do inglês, *blocking job shop* - BJS) podem ser vistas em (MOGALI *et al.*, 2021; DABAH *et al.*, 2019; LANGE; WERNER, 2019). Geralmente, os métodos para lidar com BJSs envolvem meta-heurísticas como recozimento

simulado (em inglês, *simulated annealing* - SA) e busca tabu (em inglês, *tabu search* - TS), que são aprimoradas para explorar soluções de escalonamento viáveis. Entretanto, devido ao grande número de soluções inviáveis obtidas durante a busca, é necessário combinar tais métodos com estratégias de recuperação de viabilidade. Mogali *et al.* (2021) argumentaram que a etapa de recuperação de viabilidade desacelera consideravelmente o algoritmo usado, incorrendo em uma enorme quantidade de tempo para explorar pequenas áreas do espaço de busca. Além disso, os recursos necessários para a troca de *jobs* devem ser considerados para resolver o impasse.

Como problemas de escalonamento geralmente são NP-difíceis, muitos pesquisadores se concentraram em projetar novos métodos heurísticos específicos ou aplicar métodos existentes para resolverem problemas de *job-shop* (DURASEVIć; JAKOBOVIć, 2020). Porém, a maioria dos algoritmos de JS são incapazes de identificar impasses durante a criação de cronogramas de produção. Doush *et al.* (2019) destacaram que as abordagens tradicionais, geralmente usando técnicas heurísticas para construir uma solução de escalonamento do zero, fazem isso sem nenhum processo de melhoria iterativo. Ainda que a maioria das pesquisas se concentrem no escalonamento da produção em várias frentes, abordagens para evitar impasses por meio de pré-configurações nos cronogramas de produção não foram exploradas.

Considerando a literatura revisada, fica claro que caracterizar impasses e descrever estratégias de prevenção na modelagem de um FMS é um desafio. Embora existam abordagens eficientes para minimizar a complexidade estrutural dos supervisores modelados com PN, os requisitos para a prevenção de impasses podem mudar ou exigir novas especificações em algum momento. Por exemplo, ao adicionar ou remover recursos, ao alterar a capacidade de um recurso ou ao adicionar um novo produto no sistema, o controle projetado originalmente para o FMS pode falhar. Além disso, a modelagem deve focar nos recursos compartilhados pelos processos e em como os elementos ativos do sistema interagem e colaboram entre si. Para Teymourifar *et al.* (2020), muitos dos modelos sugeridos geralmente não são aplicáveis a problemas reais de manufatura em que a capacidade de transporte ou o espaço de *buffer* são insuficientes. Semelhante aos sistemas ferroviários estudados por Sasso *et al.* (2021), a maioria dos sistemas de manufatura também apresentam restrições que impedem a troca de *job*s. Por exemplo, devido à falta de *buffer* ou de equipamentos adicionais, ou devido à mobilidade reduzida para movimentação dos *jobs* entre recursos. Nos casos em que as ações de recuperação podem ser difíceis ou caras de implementar, é adequado que os cronogramas de produção criados pelo FJS sejam ajustados com antecedência para estarem livres de determinados impasses.

Com relação ao escalonamento dinâmico, a literatura mostra que o número de métodos utilizados para construir o escalonamento é limitado. Isso ocorre devido à falta de informações disponíveis sobre o estado do sistema, que só serão conhecidas durante a sua execução. A inclusão de várias restrições de controle para tratar impasses, informações em tempo de execução sobre o estado dos *buffers*, dos produtos e demais recursos de produção, tornam as abordagens de escalonamento dinâmico complexas. Algumas soluções propostas requerem treinamento ou aprendizagem computacionalmente caros de realizar. Ainda que determinadas pesquisas obtenham resultados eficientes, as soluções são limitadas a alguns poucos cenários de produção, como em (LI *et al.*, 2021; DURASEVIć; JAKOBOVIć, 2020; UHLMANN; FRAZZON, 2018; SUN *et al.*, 2021; MESSINIS; VOSNIAKOS, 2020). Portanto, devido às questões de limitação e dificuldades explanadas sobre as abordagens revistas, reforça-se a importância de pré-ajustar os cronogramas de produção a fim de evitar certos tipos de impasses.

2.4 CONCLUSÕES DO CAPÍTULO

Este capítulo apresentou uma pesquisa exploratória sobre temas fundamentais para a tese, relacionados com sistemas flexíveis de manufatura e sistemas multiagentes. Dentre os principais problemas enfrentados com a implantação de sistemas flexíveis de manufatura, os problemas de escalonamento e de controle da produção estão entre os mais investigados por pesquisadores da área. Em geral, as abordagens propostas buscam definir sequências de operações de produção de acordo com os critérios de desempenho pretendidos, mas sem fornecer garantias de que os requisitos de fabricação serão atendidos conforme o escalonamento. Além disso, a maioria dos algoritmos de escalonamento propostos necessitam de aperfeiçoamentos, pois não são capazes de identificar e evitar impasses durante a criação dos cronogramas de produção. Há abordagens que procuram caracterizar e prevenir impasses na modelagem do sistema, mas em geral são incompatíveis com sistemas de manufatura com requisitos de flexibilidade e descentralização. Outras abordagens com viés em escalonamento dinâmico geralmente dependem de treinamentos conduzidos com especificidades do ambiente de experimentação, mas são limitadas a cenários específicos de produção. Algumas soluções ainda envolvem ações de recuperação que podem ser difíceis ou caras de implementar, ou não consideram restrições que impedem a recuperação do sistema (por exemplo, a falta de *buffers*).

Na literatura relacionada a sistemas multiagentes, quando há a integração de agentes em dispositivos ou equipamentos físicos, as abordagens propostas geralmente estão associadas

a equipamentos que possuem capacidade de mobilidade (por exemplo, robôs e drones). Como consequência, várias abordagens são inviáveis para aplicação em máquinas e equipamentos utilizados em ambientes de produção. Entretanto, as características de auto-organização, divisão de tarefas, cooperação, suporte ao encapsulamento de funcionalidades e a natureza descentralizada dos sistemas multiagentes podem ser aproveitadas para a modernização dos sistemas de manufatura. Os agentes podem ser integrados aos equipamentos do chão de fábrica para resolverem problemas de otimização, tomarem decisões e executar tarefas complexas com maior agilidade, suprindo as limitações e substituindo os controles centralizados. Além da descentralização do controle, que é uma tendência da Indústria 4.0, a capacidade de obtenção de consenso fornece estrutura para outras funcionalidades importantes. Em sistemas flexíveis de manufatura controlados por ordem, o aproveitamento do consenso multiagente para integrar soluções inteligentes de controle de impasses, garantindo flexibilidade nos processos produtivos e atendendo às restrições e urgências da produção, ainda não foi bem explorado.

As contribuições documentadas nesta tese tiveram como metas o escalonamento da produção livre de impasses, com garantias de flexibilidade e de previsibilidade nos processos produtivos. Enquanto outras pesquisas buscaram resolver problemas de impasses em tempo de produção, as abordagens propostas buscaram resolver determinados impasses antecipadamente, identificando-os e evitando-os por meio de reajustamentos nos cronogramas de produção. Restrições de *buffer* e de transporte do ambiente produtivo foram tratadas nas soluções, considerando o tipo de maquinário disponível no chão de fábrica e o roteamento entre eles. Conjuntamente, uma arquitetura multiagente especialmente adequada para sistemas flexíveis de manufatura foi fornecida para lidar com os cronogramas modificados. O controle multiagente desenvolvido garante uma visão completa do sistema nos momentos de convergência dos agentes para novos estados de informação. Esta característica foi aproveitada para validar a inserção de ordens de urgência no sistema, considerando as restrições de *buffer* e de transporte, além de evitar a sobreposição de operações em recursos de produção compartilhados. As abordagens garantem flexibilidade e previsibilidade sob certas condições do sistema, para melhor atender a questão de dimensionamento da produção colocada. Os resultados desta tese tem o viés inovador, as contribuições trataram de questões em aberto, não contempladas por outras pesquisas.

3 ESCALONAMENTO E CONTROLE MULTIAGENTE

Este capítulo apresenta um estudo preliminar sobre sistemas multiagentes, com abordagens que serviram de base para outros estudos da tese. Neste estudo, um MAS distribuído é proposto e experimentado em um FMS virtual. Os agentes encaminham os produtos para as estações de manufatura do FMS, conforme cronogramas de produção gerados por um algoritmo de *job-shop* flexível, e também aplicam métodos de controle para evitar certos tipos de impasses. Dois métodos de controle de impasses ocasionados pelo compartilhamento de recursos no ambiente virtual foram desenvolvidos e comparados. Ao final, os resultados do estudo preliminar são discutidos e o capítulo encerra com as conclusões.

3.1 CARACTERÍSTICAS DO TRABALHO

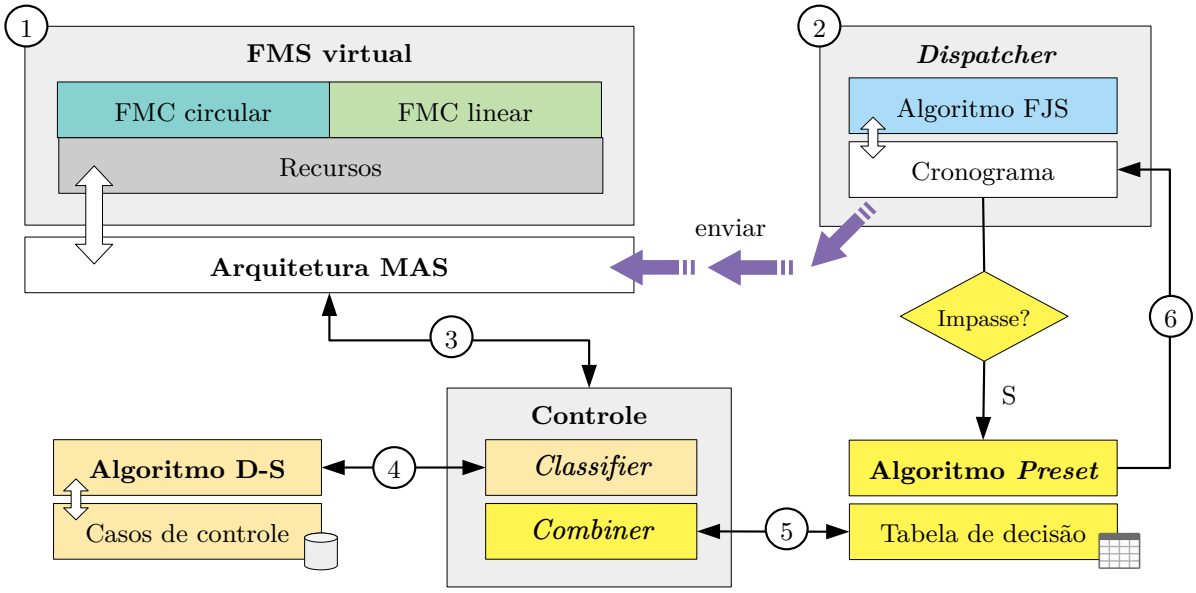
Este estudo está relacionado com os dois primeiros artigos da lista de trabalhos desenvolvidos (Seção 1.3). A motivação surgiu com a realização de pesquisas para um estudo dirigido sobre MAS, quando se identificou que a maioria das abordagens envolvendo aplicações práticas com dispositivos físicos estavam relacionadas a robôs ou outros equipamentos com capacidade de mobilidade; embora, as características dos MAS sejam apropriadas para suprir necessidades e melhorias no escalonamento e controle da produção.

Inicialmente, o estudo se concentrou no desenvolvimento de um consenso multiagente sem líderes, por ser mais adequado para a descentralização do controle e a auto-organização das operações de produção. Os agentes do MAS foram programados para controlar as operações de produção de uma *smart factory* virtual, direcionando os produtos paras as estações de manufatura definidas nos cronogramas. Um algoritmo de *flexible job-shop scheduling* convencional foi utilizado para o escalonamento. As rotas de transporte dos produtos foram incorporadas aos cronogramas para que os produtos visitassem certos grupos de máquinas em uma sequência específica. Apesar de admitir quaisquer sequências de operações, em qualquer ordem e com repetição de estações, o MAS limitou-se a executar um único produto por vez. Esta limitação evidenciou a necessidade de controles adicionais para evitar impasses (por exemplo, *deadlocks*) resultantes do compartilhamento de recursos entre diferentes operações de produção.

Para suprir as necessidades identificadas, a segunda etapa do estudo introduziu conceitos de FMS, suporte ao paralelismo de produtos e mecanismos para controle de impasses na produção.

O ambiente virtual foi atualizado para um FMS composto por dois FMCs. O problema de paralelismo foi resolvido com a inclusão de restrições de controle, considerando o acoplamento (dependência) entre agentes e a capacidade dos recursos de produção. Para prevenir potenciais impasses e assegurar flexibilidade na produção, foram propostos dois algoritmos. O primeiro algoritmo empregou uma base de casos de controle livres de impasses, com uma abordagem que habilita os agentes a escolherem o melhor caso de controle da base para conduzir a produção. Entretanto, a execução de diferentes cronogramas de produção não pôde ser garantida com programação estática e sequencial. Assim, um segundo algoritmo foi desenvolvido para operar com uma tabela de decisão aplicada no roteamento de produtos e cronogramas pré-ajustados para evitar determinados impasses no sistema.

Figura 5 – Visão geral da integração entre FMS, MAS, *dispatcher* e controles.



Fonte: Autoria própria.

Uma visão geral da abordagem proposta neste estudo é apresentada na Figura 5. Os pontos enumerados da figura são explicados a seguir:

- (1) *FMS virtual*: consiste em dois FMCs com um conjunto de recursos de fabricação composto por transportadores modulares, estações e um braço robótico.
- (2) *Dispatcher*: cria ou carrega um modelo de produção para o algoritmo FJS (escalonador) gerar o respectivo cronograma de produção. Em seguida, o cronograma regular ou pré-ajustado é enviado ao MAS.

- (3) *Arquitetura MAS*: fornece estrutura para que os agentes possam cooperar entre si, empregando consenso sem líder, e controles para conduzirem a produção livre de impasses.
- (4) *Classifier*: utiliza um algoritmo baseado na teoria de Dempster-Shafer para permitir que os agentes decidam um caso de controle apropriado para realizar o roteamento de produtos e evitar impasses (sem pré-ajustes nos cronogramas).
- (5) *Combiner*: emprega uma tabela de decisão para o roteamento da produção e pode utilizar o algoritmo *Preset* para pré-ajustes nos cronogramas preditivos.
- (6) *Algoritmo Preset*: usa os métodos *wait*, *swap* e *shift* para pré-ajustar os cronogramas de produção quando impasses forem identificados.

A arquitetura MAS é capaz de suportar a execução de diferentes planos de produção envolvendo os dois FMCs. O roteamento de produtos no sistema é realizado por transportadores modulares virtuais e braço robótico. O *dispatcher* adapta estes recursos de transporte no modelo de produção, que é submetido a um algoritmo de FJS pertencente ao pacote de software da Google OR-Tools[1]. Em geral, algoritmos de escalonamento do tipo *job-shop* não são capazes de identificar e prevenir impasses durante o escalonamento. Os algoritmos de controle propostos neste estudo podem suprir estas limitações, prevenindo certos tipos de impasses no FMS virtual.

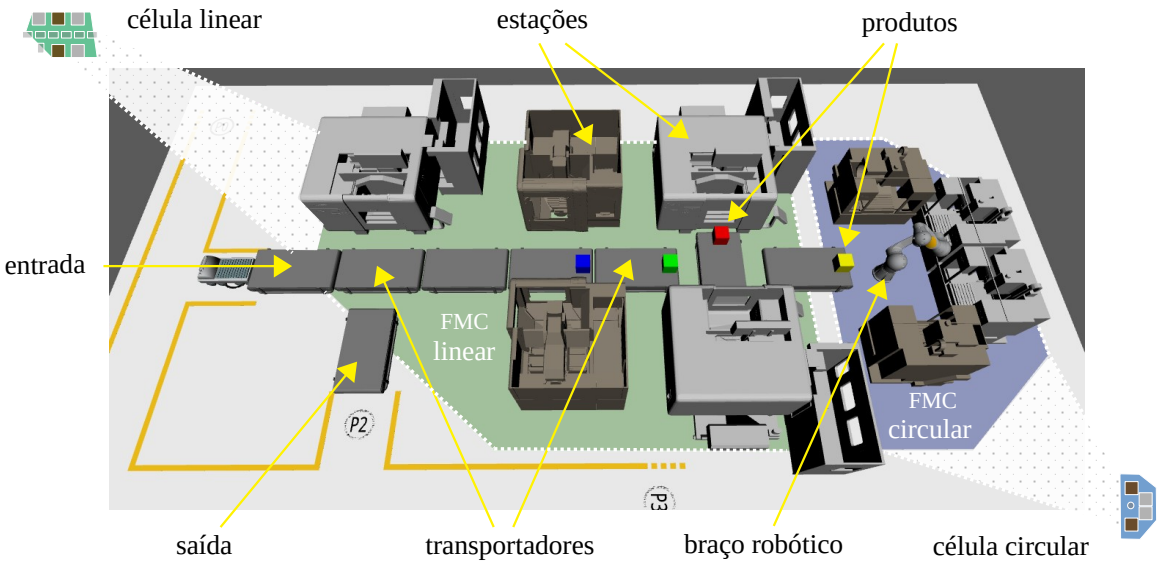
3.1.1 Estrutura do FMS virtual

O FMS virtual é uma representação gráfica personalizada, leve e interativa de um FMS fictício desenvolvido para este estudo. Foi escrito em Python, que gera as estruturas de fábrica na plataforma Rviz. Rviz é uma ferramenta 3D para ROS (*Robot Operating System*), que permite a visualização de objetos e informações utilizando *plug-ins* para diversos recursos disponíveis.

O FMS virtual, apresentado na Figura 6, é composto por um FMC linear e um FMC circular, que totalizam oito transportadores modulares ($cnv0$ a $cnv7$), nove estações de fabricação ($mst0$ a $mst8$) e um braço robótico ($arm0$). A área em verde possui cinco estações de fabricação associadas ao FMC linear. A área em azul, com quatro estações e um braço robótico, formam o FMC circular. Os produtos entram no sistema pelo transportador identificado como “entrada” ($cnv0$) e, após serem processados nas estações, saem pelo transportador “saída” ($cnv7$). O tipo de trabalho realizado pelas estações de manufatura fictícias não foi levado em consideração,

[1] Google OR-Tools (PERRON; FURNON, 2021).

Figura 6 – Divisão do FMS virtual em FMCs circular e linear.

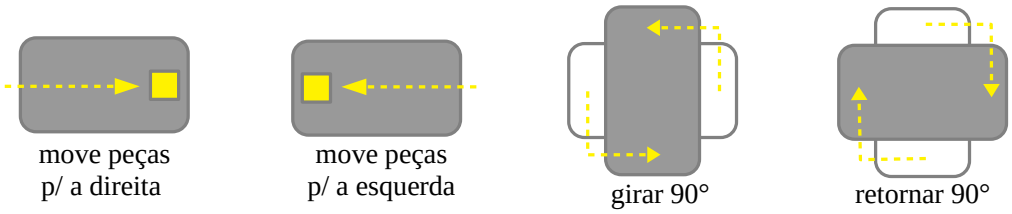


Fonte: Autoria própria.

somente a sua ocupação (livre ou em uso). Os produtos são representados no FMS por blocos coloridos, cada cor está relacionada a um tipo de produto.

No FMS virtual, os transportadores modulares possuem posições fixas e são alinhados em sequência para transportar os produtos entre as estações do FMC linear, enquanto que o braço robótico movimenta os produtos nas estações do FMC circular. Todos os transportadores podem mover produtos nas direções para frente (direita) ou para trás (esquerda) do fluxo de trabalho e girar à 90° na direção da estação de destino (Figura 7).

Figura 7 – Funcionamento dos transportadores modulares virtuais.



Fonte: Autoria própria.

Sensores virtuais foram programados nas extremidades dos transportadores para detectar a presença de produtos e indicar se o transportador está ocupado. A capacidade dos recursos (transportadores, estações e braço robótico) é de um produto por vez e o controle multiagente evita a sobrecarga. Por exemplo, os agentes não podem entregar um produto à máquina de destino enquanto outro produto estiver ocupando essa máquina. Portanto, os próprios recursos são usados

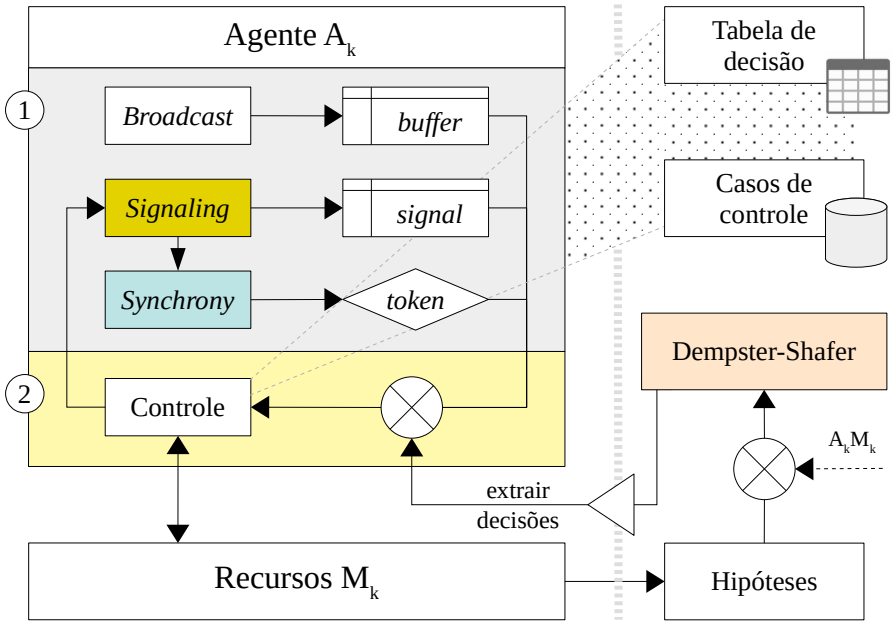
como *buffers* e retêm os produtos até que os recursos subsequentes fiquem disponíveis.

Uma *thread* exclusiva para cada agente executa a lógica de programação de seu respectivo recurso, seja ele transportador, estação de manufatura ou braço robótico. Toda a comunicação entre eles ocorre via tópicos ROS e cada agente tem sua especialização ou tarefa específica a realizar. Não há controle centralizado, cada agente processa as informações localmente dentro de seu bloco de código; o desenvolvimento dos agentes segue os princípios de um MAS distribuído. O *framework* ROS foi utilizado por ser atual e de código aberto, além de possuir ferramentas personalizáveis e possibilitar a programação de vários tipos de equipamentos.

3.1.2 Arquitetura dos agentes

A arquitetura dos agentes é dividida em duas partes: a parte de comunicação e a parte de controle (Figura 8). Na parte de comunicação, os agentes contam com suporte a ROS e utilizam três tópicos em comum para a troca de mensagens: "*broadcast*", "*synchrony*" e "*signaling*".

Figura 8 – Arquitetura dos agentes.



Fonte: Autoria própria.

O agente recebe o cronograma de produção através do tópico "*broadcast*" e armazena os eventos direcionados a ele em um *buffer* local. Com o tópico "*synchrony*", o agente atualiza o *token* (sequência de evolução do sistema) e aciona o processamento de eventos em seu sistema de controle. Por fim, no tópico "*signaling*", o agente salva uma lista dos agentes que estão processando eventos com o *token* atual; o *token* é incrementado somente se a lista estiver vazia.

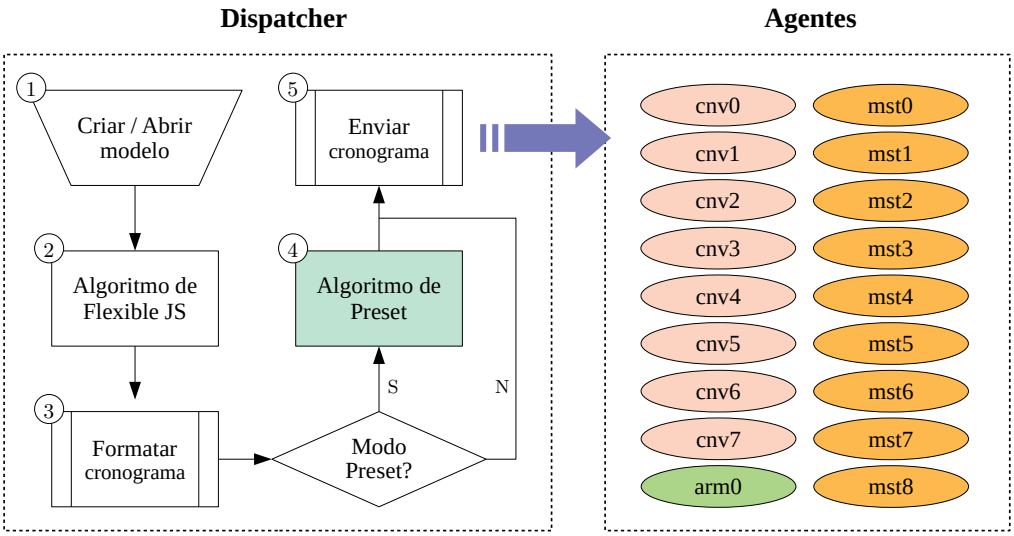
Esse esquema de troca de mensagens de sinalização e sincronização garante o consenso entre agentes para a execução de atividades compartilhadas e paralelas.

Na parte de controle, cada agente A_k está associado a um recurso de M_k no FMS e pode comandá-lo diretamente para atender o cronograma de produção (por exemplo, girar o transportador em 90°). Além disso, os agentes podem aceitar solicitações de serviço de outros agentes para alterar o estado de um recurso (por exemplo, a direção de rotação do transportador vizinho). Dois controles adicionais são utilizados para guiar as ações dos agentes. O primeiro controle permite que os agentes sigam as instruções de um caso de controle livre de impasses, indicado por um algoritmo baseado na teoria de Dempster-Shafer. O segundo algoritmo possibilita que os agentes utilizem uma tabela de decisão para o roteamento da produção. Além do roteamento, os cronogramas também podem ser pré-ajustados pelo *dispatcher* sempre que potenciais impasses forem identificados.

3.1.3 Funcionamento do *dispatcher*

O escalonamento da produção é definido por um nó ROS chamado "*dispatcher*", que executa o algoritmo de FJS e possui uma interface virtual para lidar com os cronogramas de produção. A Figura 9 apresenta uma visão geral dos processos executados pelo *dispatcher*.

Figura 9 – Processos do *dispatcher* para envio de cronogramas.



Fonte: Autoria própria.

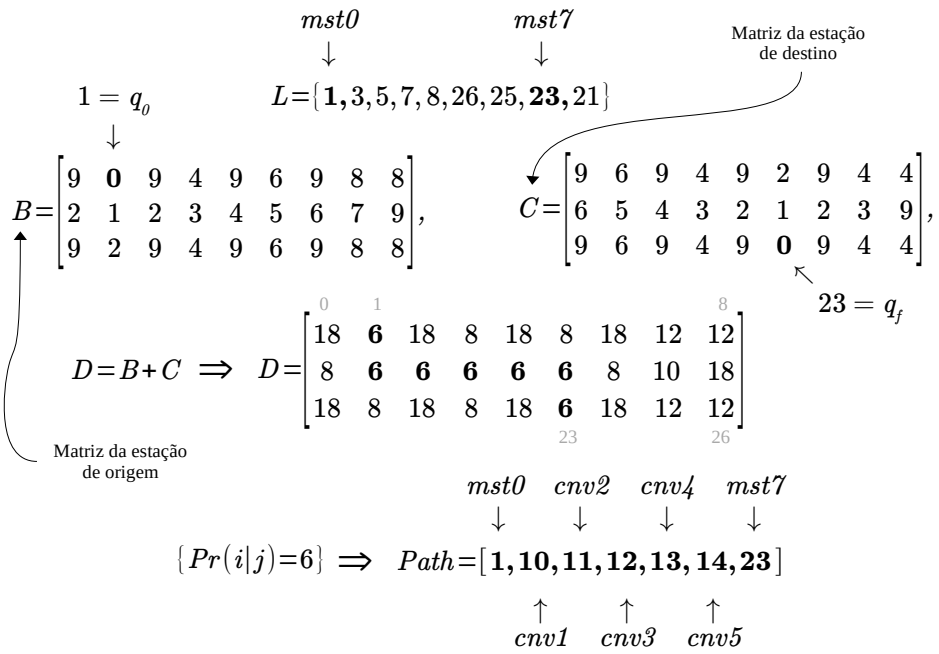
O *dispatcher* configura um cronograma de produção em cinco processos: (1) criar ou carregar os requisitos de fabricação (modelo); (2) processar os requisitos usando o algoritmo FJS;

(3) formatar o cronograma de produção; (4) pré-ajustar o cronograma de produção, se necessário (e se habilitado); (5) distribuir o cronograma para os agentes do MAS.

Dado um conjunto de agentes $A = \{A_1, A_2, ..., A_k, ..., A_m\}$, para cada A_k existe um respectivo recurso M_k da fábrica (por exemplo, transportador ou estação), tal que $M = \{M_1, M_2, ..., M_k, ..., M_m\}$. Os produtos são processados em um ou mais recursos de M, de acordo com uma sequência de operações e tempos de processamento predefinidos. O *dispatcher* inclui os transportadores modulares e o braço robótico em M, para que o escalonamento contemple o roteamento dos produtos no FMS. O processo de cálculo das rotas é caracterizado por um conjunto de localizações $L = \{L_0, L_1, ..., L_q\}$, tal que cada L_p aponta para a localização de um recurso (estação de manufatura, transportador ou braço robótico) de M, definidos em uma matriz de distância. Cada recurso tem sua própria matriz de distância; elas são combinadas para o cálculo das rotas de acordo com uma localização inicial e final (q_0, q_f) do espaço L.

Uma matriz de distância $D = (d_{ij})_{mxn}$ atua como uma cadeia de Markov sobre um espaço de probabilidade finito. Portanto, $Pr(i \mid j) = d_{ij}$ representa a probabilidade de mover i ou j na matriz alterando a posição atual um passo de cada vez, até d_{ij} indicar q_f. Na prática, cada estação de manufatura está associada a uma matriz de distância para garantir que a célula que representa a localização da estação na matriz tenha o valor $q_0 = 0$, enquanto que as células restantes são incrementadas em 1 (para cada célula a partir de q_0). Então, o caminho entre duas matrizes B e C é calculado por $B + C = (b_{ij} + c_{ij})_{mxn}$, onde $0 <= i < m$ e $0 <= j < n$.

Exemplo 1: Processo de cálculo de rota com matrizes de distância.



Um exemplo de cálculo de rota é apresentado (Exemplo 1). A matriz de distância $D_{3\times9}$ tem 27 valores de custo (ou distância). Cada valor em $Path$ representa um índice em D, que indica a localização do menor valor de custo calculado ($d_{ij} = 6$). A sequência de índices de *Path* equivalem ao caminho entre as estações de fabricação $mst0$ (q_0) e $mst7$ (q_f). Este processo é realizado para cada par de estações ou de uma estação até um transportador, de forma a representar um segmento de rota para o produto. O conjunto de rotas e de estações representam os requisitos de fabricação, um "modelo" para o *dispatcher* gerar o cronograma de produção.

O Exemplo 2 apresenta um modelo de produção utilizado pelo *dispatcher* para gerar o respectivo cronograma. O tempo de processamento para as estações de manufatura (identificadas de 8 até 16) é definido como 5 para simplificar o exemplo; para os transportadores (de 0 até 7) e para o braço robótico (17), este tempo é definido como 1.

Exemplo 2: Modelo de fabricação para o *dispatcher*.

cnv0 *cnv1* *mst0* *cnv0* *cnv2* *cnv3* *mst8* ...
↓ ↓ ↓ ↓ ↓ ↓ ↓

$Modelo =$ [[(0, 1), (1, 1), (**8**, 5), (1, 1), (2, 1), (3, 1), (**16**, 5), (3, 1), (2, 1), (1, 1), (7, 1)], [(0, 1), (1, 1), (**8**, 5), (1, 1), (2, 1), (3, 1), (**9**, 5), (3, 1), (2, 1), (1, 1), (7, 1)], [(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (17, 1), (**11**, 5), (17, 1), (**14**, 5), (17, 1), (6, 1), (5, 1), (4, 1), (3, 1), (2, 1), (1, 1), (7, 1)],[(0, 1), (1, 1), (2, 1), (3, 1), (**9**, 5), (3, 1), (4, 1), (5, 1), (**10**, 5), (5, 1), (4, 1), (3, 1), (2, 1), (1, 1), (**8**, 5), (1, 1), (7, 1)]]

Cada recursos é representado no modelo por um *id* e um *tempo* de processamento em segundos (por exemplo, *mst8* = (16, 5)). O modelo inclui requisitos para quatro produtos: $j_0 = [mst0, mst8]$, $j_1 = [mst0, mst1]$, $j_2 = [mst3, mst6]$ e $j_3 = [mst1, mst2, mst0]$. Em seguida, o modelo é submetido ao algoritmo FJS para obter o respectivo cronograma de produção. Posteriormente, o cronograma é formatado em ordem crescente de tempo e um índice de sequência (*token*) é atribuído a cada evento agendado. Por fim, o algoritmo de *Preset* (para pré-ajustes no cronograma) analisa a presença de eventos causadores de impasses e, em caso afirmativo, realiza ajustes antes que o cronograma seja distribuído para os agentes.

3.1.4 Cronograma de produção

Um cronograma de produção é composto por f eventos, tal que $E = \{E_0, E_1, ..., E_e, ..., E_f\}$. Cada evento representa uma tarefa que deve ser realizada por um agente

específico. O *dispatcher* distribui o cronograma, evento por evento, para todos os agentes MAS. Um evento $E_e = [start, duration, agent, job, token]$ possui cinco atributos, descritos a seguir:

- $start$: tempo de início do evento.
- $duration$: tempo de duração do evento.
- $agent$: identifica o número do agente que controla o evento.
- job: identifica o produto que compõe o evento.
- $token$: índice de sequência para controle da lista de eventos.

Os atributos $start$ e $duration$ são dados em segundos. Por padrão é definido $start = 0$ para o evento E_0, que é o primeiro evento da lista de eventos de um cronograma de produção. O atributo $agent$ identifica o agente que deve processar o evento (de 0 a 7 para os transportadores $cnv0$ à $cnv7$; de 8 a 16 para as estações $mst0$ à $mst8$; e 17 para o braço robótico $arm0$). O atributo job identifica qual é o produto bruto que possui relação com o evento, a fabricação do produto é definida por um subconjunto de eventos de E. O atributo $token$ refere-se a um índice de sequência que determina quais eventos podem ser processados em um determinado momento, para garantir o processamento paralelo de operações nos agentes que detém o *token* atual.

Um cronograma de produção é formado por uma sequência de eventos. No cronograma do Exemplo 3, os agentes 1, 3 e 9 poderão processar seus respectivos eventos de *token* igual a 4 em paralelo. Já o evento que possui *token* igual a 6, será processado apenas pelo agente 5.

Exemplo 3: Exemplo de parte de um cronograma.

start $\longrightarrow$ $[0\text{L}, 1, 0, 3, 0]$

$\vdots$

$[4\text{L}, 1, 1, 1, 4]$
$[4\text{L}, 1, 3, 2, 4]$ $\longleftarrow$ *eventos em paralelo*
$[4\text{L}, 5, 9, 3, 4]$
$[5\text{L}, 1, 4, 2, 5]$
$[5\text{L}, 5, 8, 1, 5]$
$[6\text{L}, 1, 5, 2, 6]$ $\longleftarrow$ *evento individual*
$[7\text{L}, 1, 6, 2, 7]$

$\vdots$

$[28\text{L}, 1, 7, 3, 28]$

O mecanismo de consenso é projetado para incrementar o *token* somente quando todos os agentes concluírem suas operações no *token* atual. Para garantir sincronismo no sistema, os

agentes que processam *jobs* com tempos mais curtos devem aguardar pelos agentes que processam *jobs* mais longos, para que o *token* seja modificado. Este é o princípio de funcionamento do mecanismo de consenso proposto.

3.2 CONTROLE DE IMPASSES

Esta seção apresenta os algoritmos propostos para lidar com problemas de impasses no FMS virtual. O objetivo dos algoritmos é garantir a completa execução dos cronogramas, impedindo que o sistema evolua para estados indesejáveis e propensos a bloqueios. Os algoritmos são associados a dois métodos de controle implementados no MAS. Desta forma, os agentes podem operar a produção utilizando cronogramas pré-ajustados ou casos de controle livres de impasses, sem a necessidade de aplicar métodos de reescalonamento em *runtime*.

3.2.1 Algoritmo *Classifier*

O algoritmo *Classifier* tem como base a teoria de Dempster-Shafer, também chamada de teoria da evidência. A teoria de Dempster-Shafer (D-S) é uma teoria matemática para modelar incertezas epistêmicas. Ela possibilita combinar evidências de diferentes fontes, para chegar a um grau de credibilidade indicado por uma função de crença. Sendo Θ um conjunto de n eventos (ou hipóteses) mutuamente exclusivos e coletivamente exaustivos, indicados por $\Theta = \{\theta_1, \theta_2, ..., \theta_i, ..., \theta_n\}$, é denominado como "quadro de discernimento". O conjunto de potência de Θ é indicado por 2^Θ, a saber $2^\Theta = \{\phi, \{\theta_1\}, ..., \{\theta_n\}, \{\theta_1, \theta_{q2}\}, ..., \{\theta_1, \theta_2, ..., \theta_i\}, ..., \Theta\}$, onde os elementos de 2^Θ ou subconjuntos de Θ são chamados de proposições (ou evidências). A estrutura D-S permite que a crença sobre tais proposições seja representada como intervalos, limitados por dois valores: crença (limite inferior) e plausibilidade (limite superior).

De acordo com Shafer (1976), probabilidades subjetivas (ou "massas") devem ser atribuídas a todos os subconjuntos de 2^Θ; ou seja, é realizado um mapeamento da massa μ de 2^Θ nos intervalos $[0, 1]$, formalmente definido por: $\mu : 2^\Theta \rightarrow [0, 1]$. A crença mede a força da evidência a favor de uma proposição P, sendo 0 a indicação de nenhuma evidência a 1 denotando certeza. A plausibilidade é a quantidade máxima de certeza que pode ser atribuída a uma proposição P (subconjunto de 2^Θ). Formalmente as funções de crença e plausibilidade são indicadas por $Bel(P)$ e $Pl(P)$. Assim, a função de crença $Bel(P)$ indica a que ponto as informações fornecidas por uma fonte sustentam P. Enquanto que a função de plausibilidade

$Pl(P)$, indica a que ponto as informações fornecidas por uma fonte não contradizem P.

Segundo Han *et al.* (2019), a D-S é amplamente aplicada para fusão de dados em processos de produção devido ao seu excelente desempenho em situações de incerteza. A incerteza é condição comum em um MAS distribuído, já que os agentes geralmente possuem conhecimento limitado do estado global do sistema. Portanto, a habilidade de lidar com incertezas é fundamental para que um agente possa interagir no FMS de maneira inteligente, escolhendo estratégias de controle adequadas diante de impasses e outros imprevistos. De acordo com Reineking (2014), um agente com uma representação explícita da incerteza sabe as limitações de seu conhecimento e pode, portanto, prever melhor os resultados de suas ações.

O *Classifier* faz a ponte entre a D-S e os agentes, habilitando-os a decidirem sobre os melhores casos de controle livres de impasses para a execução dos cronogramas de produção. Os agentes executam esse processo de decisão em três etapas. A primeira etapa (Expressão 1), que correspondendo à extração de proposições sobre um cronograma de produção E, é definida por

$$\bigcup_{\rho=0}^{|E|} H_\rho\{A_k, J_i, E_{e[token]}\}, \tag{1}$$
$$\forall k \in A \mid 8 \leq k \leq 16, \forall i \in J,$$
$$\forall e \in E \mid E_e \rightarrow E_{e[token]}$$

onde H é um conjunto de ρ subconjuntos de proposições extraídas de E (do *buffer* de cada agente A_k, especificamente), $|E|$ é o tamanho do cronograma, i identifica o *job* de J que possui relação com o evento, $E_{e[token]}$ corresponde ao índice de sequência do evento (o 5º atributo de E_e), e k identifica o agente associado à estação de manufatura ($8 \leq k \leq 16 \rightarrow [mst0, ..., mst8]$).

$$\begin{aligned} K = \{&\alpha_0[(\{A_{k_{\alpha_0}}, J_{i_{\alpha_0}}, E_{e[token]_{\alpha_0}}\}, \sigma_{0_{\alpha_0}}), ..., (..., \sigma_{\lambda_{\alpha_0}})], \\ &\alpha_1[(\{A_{k_{\alpha_1}}, J_{i_{\alpha_1}}, E_{e[token]_{\alpha_1}}\}, \sigma_{0_{\alpha_1}}), ..., (..., \sigma_{\lambda_{\alpha_1}})], \\ &\vdots \\ &\alpha_{|K|}[(\{A_{k_{\alpha_{|K|}}}, J_{i_{\alpha_{|K|}}}, E_{e[token]_{\alpha_{|K|}}}\}, \sigma_{0_{\alpha_{|K|}}}), ..., (..., \sigma_{\lambda_{\alpha_{|K|}}})]\} \end{aligned} \tag{2}$$

Dado um quadro de discernimento definido por $\Theta = \{A, J, E\}$, tal que A é o conjunto de agentes, J é o conjunto de *jobs* e E é o conjunto de eventos do cronograma de produção, o

conjunto de massas relevantes para as operações executadas no FMS é denotado por $K : K \subset \{2^{\Theta} \rightarrow [0,1]\}$ (Equação 2).

A segunda etapa (Equação 3) do processo de decisão envolve a obtenção dos valores de crença em K, para o conjunto de proposições H (ou hipóteses), que é dado por

$$S = \sum_{x} \sum_{\alpha=0}^{|K|} (Bel(H) \geq \omega, K_{\alpha}), \quad \forall \{H \mid H \subseteq \Theta, Bel(H) \geq \omega\}, \quad \forall \alpha \in K, \omega = 0.7 \text{ def.} \tag{3}$$

onde S é o conjunto de x valores de crença e índices de massa (δ, α) que satisfazem ω, Bel é a função de crença sobre H para cada massa K_{α}, ω é a constante limitadora de crenças pouco claras ($\delta < 0{,}7$), e $|K|$ é a cardinalidade do conjunto de massas K.

O conjunto de casos de controle é denotado por C, tal que C garante a mesma cardinalidade que K. Portanto, para cada K_{α} existe um caso de controle específico C_{α}. Assim, a terceira etapa do processo decisório envolve selecionar, a partir do conjunto de x crenças e $S\{(\delta,\alpha)\}$ índices, o melhor caso de controle indicado para orientar os agentes na execução do cronograma de produção atual.

O cálculo do melhor caso de controle (Expressão 4) é definido por

$$max(S) \rightarrow C_{\alpha}, \forall (\delta,\alpha) \in S \mid S_{x[\alpha]} \rightarrow C_{\alpha} \tag{4}$$

onde, $max(S)$ indica o maior valor de crença $S_{x[\delta]}$ e respectivo índice de massa $S_{x[\alpha]}$, tal que $S_{x[\alpha]}$ é índice de um caso de controle em C.

O pseudocódigo do *Classifier* é apresentado em Algoritmo 1. A, J, E e K são respectivamente os conjuntos de agentes, tarefas, eventos e massa. O procedimento GET_HYPOTHESIS corresponde ao primeiro passo do processo de decisão executado pelos agentes; GET_BELIEFS corresponde ao segundo passo, e SET_CASE ao terceiro. Cada agente do MAS é responsável por executar as etapas do processo de decisão pelo menos uma vez durante a execução do FMS. Uma vez que um caso de controle de C é indicado, cada agente passa a usar as instruções deste caso de controle livre de impasses para executar o cronograma de produção.

Algorithm 1 *Classifier*

$A \leftarrow \{A_k, ..., A_m\}$, ▷ conjunto de agentes, $jobs$, eventos e massa
$J \leftarrow \{J_i, ..., J_n\}, E \leftarrow \{E_e, ..., E_f\}$,
$K \leftarrow \{\alpha_0[(\{A_{k_{\alpha_0}}, J_{i_{\alpha_0}} E_{e[token]_{\alpha_0}}\}, \sigma_{\lambda_{\alpha_0}})], ..., \alpha_{|K|}[...]\}$

procedure GET_HYPOTHESIS(A, J, E) ▷ (passo 1)
  for $\rho \leftarrow 0, |E|$ **do**
    $H_p \leftarrow \{A_k, J_i, E_{e[token]}\}$ ▷ (conjunto de hipóteses)
  end for
end procedure

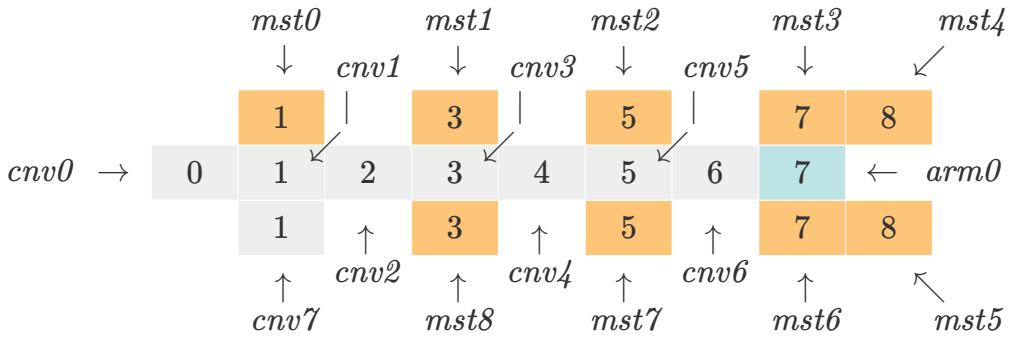
procedure GET_BELIEFS(H, K) ▷ (passo 2)
  $\omega \leftarrow 0.7, x \leftarrow 0$
  for $\alpha \leftarrow 0, |K|$ **do**
    $\delta \leftarrow K_\alpha.Bel(H)$ ▷ função de crença do D-S
    if $\delta \geq \omega$ **then** ▷ valida crença na constante limitadora
      $S_x \leftarrow (\delta, \alpha)$
      $x \leftarrow x + 1$
    end if
  end for
end procedure

procedure SET_CASE(S) ▷ (passo 3)
  $max \leftarrow 0$
  for $x \leftarrow 0, |S|$ **do**
    if $S_{x+1[\delta]} > S_{x[\delta]}$ **then**
      $max \leftarrow S_{x+1[\alpha]}$
    end if
  end for
  $C_\alpha \leftarrow max$ ▷ decisão indicada
end procedure

3.2.2 Algoritmos *Combiner* e *Preset*

Em vez utilizar instruções contidas em casos de controle, o algoritmo denominado "*Combiner*" orienta o MAS no roteamento de produtos no FMS empregando uma tabela de decisão (Tabela 3). As células em cinza representam os transportadores, em azul o braço robótico e em marrom as estações de manufatura.

Tabela 3 – Tabela de decisão para roteamento de produtos com o Combiner.



Fonte: Autoria própria.

A tabela de decisão indica valores de inferência para cada recurso do sistema e a relação de vizinhança entre eles. Os agentes tomam decisões com base na localização atual

(CL) de um produto na tabela e a sua próxima localização (NL). O *Combiner* obtém CL e NL, respectivamente, do evento corrente E_e e do próximo evento $E_{e+\eta}$ de mesmo job_id que o evento corrente, de acordo com o cronograma de produção.

A decisão de movimentar um produto J_i no FMS é executada com base em quatro premissas (Equação 5), dadas por

$$move(J_i) = \begin{cases} front & \forall CL < NL, \\ back & \forall CL > NL, \\ up & \forall CL = NL, \forall r \mid r1 > r2, \\ down & \forall CL = NL, \forall r \mid r1 < r2 \end{cases} \tag{5}$$

$$\forall (CL, NL) \mid \{CL \rightarrow (E_e, r1), NL \rightarrow (E_{e+\eta}, r2), 0 \leq (r1, r2) \leq 2\},$$
$$\forall E_{e[job]} = E_{e+\eta[job]}, \forall i \in J \mid i \rightarrow E_{e[job]}, \eta \leq |E|$$

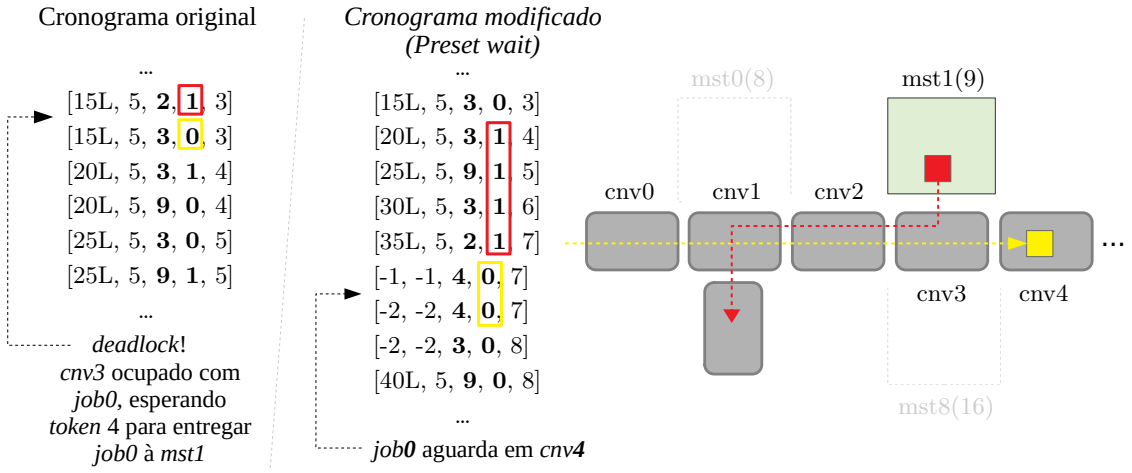
onde $move(J_i)$ é a função de mover um produto $i \in J$ no FMS; "$front$", "$back$", "up" e "$down$" indicam as direções do movimento, as comparações entre CL e NL representam as premissas, $r1$ e $r2$ indicam respectivamente as linhas da tabela para CL e NL, e $|E|$ é a cardinalidade do cronograma E.

O *Combiner* pode ser usado em conjunto com "*Preset*", um algoritmo de pré-ajustes do cronograma de produção. O algoritmo *Preset* procura evitar que determinados impasses ocorram no cronograma de produção gerado pelo algoritmo FJS. Se forem identificadas combinações de eventos que potencialmente levam o sistema à situações de impasses, o *Preset* tentará modificar o cronograma de produção antes que ele seja distribuído para os agentes do MAS. O algoritmo *Preset* é capaz de executar até três tipos de pré-ajustes no cronograma de produção:

- *Wait*: mantém determinados eventos de um *job* em espera até que o recurso ocupado por outro *job* em processamento seja liberado, evitando assim um possível *deadlock*.
- *Swap*: troca a posição de um evento no cronograma, a partir de um determinado índice (*token*), quando um potencial impasse é identificado.
- *Shift*: desloca todos os eventos de mesmo *job* em até um *token* de incremento, para evitar um potencial *deadlock* identificado no cronograma.

Exemplos de funcionamento do algoritmo *Preset* são apresentados a seguir. No Exemplo 4, o algoritmo *Combiner* não pode lidar com um impasse identificado no cronograma gerado pelo FJS, onde os *jobs* J_0 e J_1 (em amarelo e em vermelho) concorrem pelo uso da estação $mst1$ (identificada pelo número de agente 9, entre parênteses). O problema ocorre nos eventos de *token* igual a 3, porque o *job* J_0 permanece em $cnv3$ e bloqueia o recurso enquanto o *job* J_1 está em $cnv2$ (agente 2) e aguardando que $cnv3$ (agente 3) seja liberado. Assim, o *token* não é incrementado para os próximos eventos devido a uma situação de *deadlock*.

Exemplo 4: Funcionamento do *Preset wait* em um caso de *deadlock*.



Usando o método *wait*, o algoritmo *Preset* faz com que o *job* J_0 avance para o próximo transportador ($cnv4$), localizado imediatamente após $cnv3$, para aguardar até que o *job* em processamento (J_1) libere $mst1$. Em seguida, o algoritmo *Preset* inclui eventos adicionais no cronograma (aqueles caracterizados por sinais negativos) que modificam a rota do produto.

Algorithm 2 *Preset wait*

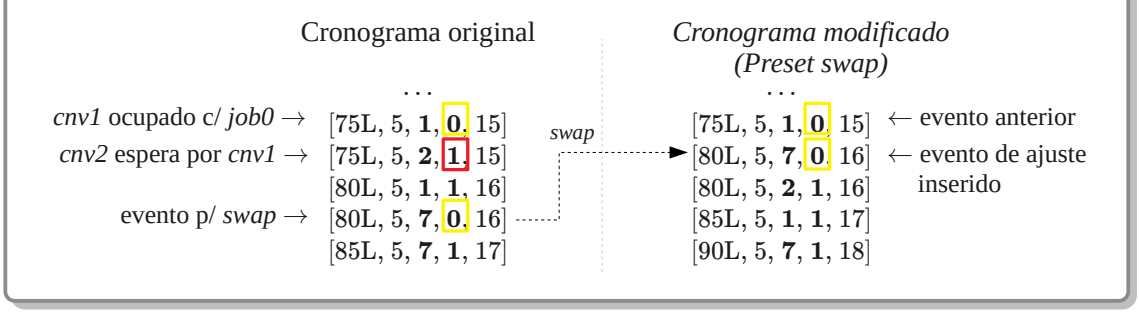
```
1:  E ← {E_e, ..., E_f}                                   ▷ conjunto de eventos
2:  function PRESET_WAIT(E)
3:      if checkLock(E) then
4:          e ← blockPoint(E)                             ▷ procura ponto de bloqueio
5:          E', E'' ← split(E, e − 1)                     ▷ divide E no ponto e − 1
6:          E ← E' + {W_0, ..., W_n} + E''                ▷ cria novo E incluindo os eventos de espera
7:          for i ← e, |E| do
8:              E_i ← adjustE(E_i[start], E_i[token])     ▷ ajusta valores de start e token
9:          end for
10:     end if
11:     return E
12: end function
```

O pseudocódigo do *Preset wait* é apresentado no Algoritmo 2. Primeiro, o algoritmo verifica o cronograma ($checkLock$) e identifica o evento que causa o bloqueio ($blockPoint$). Em

seguida, o cronograma é dividido em dois subconjuntos ($E^{'}$ e $E^{''}$), no ponto anterior ao ponto de bloqueio ($e-1$) para incluir os eventos de espera W. Os valores de *start* e $token$ são ajustados no novo cronograma a partir do ponto de bloqueio.

No Exemplo 5, o cronograma gerado pelo algoritmo FJS resultará em *deadlock* por espera circular, nos agentes $cnv1$ e $cnv2$ (agentes 1 e 2, respectivamente), para os eventos de *token* igual a 15. No método *swap*, o *Preset* mudará a posição do evento $[80L, 5, 7, 0, 16]$ para após o evento anterior de mesmo *job_id* e ajustará os tempos e *tokens* no restante do cronograma de produção, eliminando o impasse. Em alguns casos, o *makespan* poderá aumentar (o tempo do evento que foi trocado, no máximo), se comparado ao cronograma original; entretanto o sistema permanecerá em operação.

Exemplo 5: Funcionamento do *Preset swap* em um caso de *deadlock*.



O pseudocódigo do *Preset swap* é apresentado a seguir (Algoritmo 3). O algoritmo verifica se existe bloqueio no cronograma ($checkLock$) e identifica o evento que pode liberar o bloqueio ($eventUnlock$). Após, é localizado o evento anterior (μ) de mesmo job_id que o evento que libera o bloqueio (e), para então trocar sua posição ($swapEvent$). Em seguida, os valores de $start$ e $token$ são ajustados no novo cronograma a partir do ponto de localização subsequente ao evento anterior ($swapEvent + 1$).

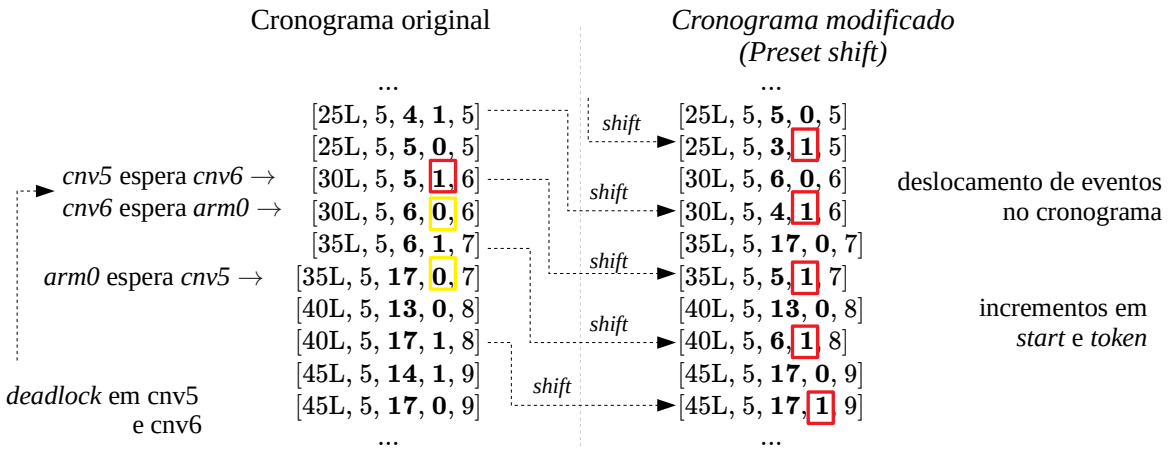
Algorithm 3 *Preset swap*

```
1:  E ← {E_e, ..., E_f}                                   ▷ conjunto de eventos
2:  function PRESET_SWAP(E)
3:      if checkLock(E) then
4:          e ← eventUnlock(E)                            ▷ encontra evento de desbloqueio
5:          μ ← previousJob(E_e)                          ▷ evento anterior de mesmo job_id
6:          E ← swapEvent(E, e, μ)
7:          for i ← μ + 1,|E| do
8:              E_i ← adjustE(E_i[start], E_i[token])     ▷ ajusta start e token
9:          end for
10:     end if
11:     return E
12: end function
```

No Exemplo 6, também há problemas de *deadlock* identificados pelo *Preset* nos eventos

com *token* igual a 6 (eventos $[30L, 5, 5, 1, 6]$ e $[30L, 5, 6, 0, 6]$). O problema ocorre porque $cnv6$ (agente 6) permanece ocupado com o *job* J_0 até que $arm0$ (agente 17) retire o *job*. No entanto, o *token* para $arm0$ só poderá ser incrementado quando $cnv5$ (agente 5) entregar o *job* J_1 para $cnv6$ (que permanece ocupado). Um dos fatores que fazem com que os transportadores e estações permaneçam ocupados com um produto é o *buffer* limitado.

Exemplo 6: Funcionamento do *Preset shift* em um caso de *deadlock*.



O *Preset* emprega o método *shift* para deslocar os eventos de mesmo *job_id* que o evento que está causando o bloqueio. Além disso, todos os eventos deslocados são incrementados em um *token* de sincronismo para evitar o bloqueio. Nesses casos, o aumento de *makespan* no cronograma de produção modificado é inevitável, porque, em relação ao cronograma original, um conjunto de eventos do mesmo produto é deslocado de posição. No entanto, o deslocamento evita *deadlocks* que poderiam ocorrer no cronograma original.

Algorithm 4 *Preset shift*

```
1:  E ← {E_e, ..., E_f}                                  ▷ conjunto de eventos
2:  function PRESET_SHIFT(E)
3:      if checkLock(E) then
4:          e ← blockPoint(E)                            ▷ procura ponto de bloqueio
5:          v[0..n] ← indexJ(E, E_e[job])                ▷ índices de E de mesmo job_id
6:          E ← shiftEvents(E, e, v)
7:          for i ← 0,|E| do
8:              E_i ← adjustE(E_i[start], E_i[token])    ▷ ajusta valores de start e token
9:          end for
10:     end if
11:     return E
12: end function
```

O pseudocódigo do *Preset shift* é apresentado a seguir (Algoritmo 4). Semelhante ao *Preset wait*, o algoritmo *Preset shift* verifica o cronograma ($checkLock$) e identifica o evento que está causando o bloqueio ($blockPoint$). Em seguida, é gerada uma lista de índices dos

eventos de mesmo *job_id* que o evento bloqueante ($indexJ$). Os eventos na lista são deslocados ($shiftEvents$) avançando em uma posição de *token* no cronograma. Por fim, os valores $start$ e $token$ de todos os eventos do cronograma de produção modificado são corrigidos.

3.3 RESULTADOS E DISCUSSÃO

Um conjunto de 32 cronogramas de produção variando entre 10 a 47 eventos foram usados como *benchmark* para avaliar os algoritmos *Classifier* e *Combiner*. Os experimentos foram conduzidos em um computador Intel Core i5-8250U 1.60GHz com 8GB de RAM. Um material complementar com os cronogramas e diagramas de Gantt gerados para os experimentos, e um vídeo mostrando o funcionamento do FMS virtual está disponível online[2]. Os resultados dos experimentos estão em cinco tabelas, que exibem o número de eventos por cronograma, o comprimento ótimo calculado pelo FJS (*length*, em segundos), número de agentes, algoritmos executados ("CSC" para *Classifier* e "CBC" para *Combiner*), memória física ocupada pelo processo (*Resident Set Size* - RSS, em *kilobytes*), percentual de CPU e de memória (MEM), e tempo gasto com a conclusão dos trabalhos (*makespan*, em segundos); os dados de RSS, MEM, CPU e Makespan são compostos por médias. A biblioteca *psutil*[3] foi utilizada para recuperar informações sobre processos em execução e utilização no sistema.

A Tabela 4 apresenta os resultados de comparação entre *Classifier* e *Combiner* (sem *Preset*), para 10 diferentes cronogramas de produção que variam entre 14 e 30 eventos. Os resultados marcados com "*approved*" (✓), para as colunas CSC e CBC, indicam que os cronogramas foram concluídos sem impasses. "*Length*" refere-se à soma dos tempos gastos com a movimentação dos produtos através dos transportadores e braço robótico, que são somados aos tempos de máquina de cada produto no cronograma de produção. O algoritmo FJS calcula tempos de 5 segundos para cada transportador no trajeto do produto e 10 segundos para cada ação do braço robótico (quando participa do trajeto). Estes valores de tempo são estimados por observação e, portanto, diferem do *makespan* (tempo real de conclusão do cronograma).

Na prática, o *makespan* está atrelado ao deslocamento de objetos virtuais no Rviz, que representam os produtos. Este deslocamento foi programado com um *delay* de 1 milissegundo para cada mudança de posição ($<$ 1,0mm em x,y) dos objetos virtuais, além dos *delays* que representam as operações de produção nas estações de manufatura. Entretanto, de acordo com a

[2] Material complementar: https://github.com/alexlds77/asc2021; Vídeo: https://youtu.be/HVb7o6DKsT0.
[3] Psutil: https://pypi.org/project/psutil/.

Tabela 4 – Comparação entre os algoritmos *Classifier* e *Combiner* (sem *Preset*).

	Eventos	***Length* (sec.)**	**Agentes**	**CSC**	**CBC**	***Wait***	***Swap***	***Shift***	**RSS (KB)**	**CPU (%)**	**MEM (%)**	***Makespan* (sec.)**
01	14	45	7	✓	-	-	-	-	195.00	150.86	2.52	47.05
	14	45	7	-	✓	-	-	-	194.93	152.10	2.52	45.18
02	14	45	7	✓	-	-	-	-	195.24	156.16	2.53	43.75
	14	45	7	-	✓	-	-	-	194.63	156.12	2.52	43.90
03	22	65	9	✓	-	-	-	-	195.30	154.87	2.53	67.22
	22	65	9	-	✓	-	-	-	195.01	154.06	2.52	64.19
04	22	65	9	✓	-	-	-	-	195.11	153.74	2.53	67.37
	22	65	9	-	✓	-	-	-	194.53	155.00	2.52	63.05
05	22	65	9	✓	-	-	-	-	195.33	153.32	2.53	68.96
	22	65	9	-	✓	-	-	-	194.82	156.00	2.52	64.41
06	22	65	9	✓	-	-	-	-	195.30	157.56	2.53	64.87
	22	65	9	-	✓	-	-	-	195.10	155.95	2.53	63.09
07	18	65	7	✓	-	-	-	-	200.22	150.16	2.59	62.32
	18	65	7	-	✓	-	-	-	200.57	156.43	2.60	58.53
08	18	65	9	✓	-	-	-	-	200.75	157.14	2.60	63.31
	18	65	9	-	✓	-	-	-	199.97	154.92	2.59	61.99
09	22	95	11	✓	-	-	-	-	199.64	151.90	2.58	96.97
	22	95	11	-	✓	-	-	-	201.40	153.40	2.61	92.55
10	30	95	11	✓	-	-	-	-	200.63	157.40	2.60	94.83
	30	95	11	-	✓	-	-	-	201.09	159.73	2.60	99.48

Fonte: Autoria própria.

Tabela 4, é possível observar que o *Combiner* reduz o *makespan* e é mais ágil que o *Classifier* em quase todos os cronogramas. Uma exceção é o caso de controle para o cronograma 10, que "força antecipadamente" o incremento de *token* para ativar o próximo evento e impedir que um produto fique ocioso no transportador $cnv6$, caso o braço robótico esteja livre. Entretanto, esse tipo de otimização é difícil de implementar e é específico para cada caso de controle.

Ainda, de acordo com a Tabela 4, o percentual de memória geral ocupada (MEM) é semelhante em quase todos os cronogramas. No entanto, a quantidade de memória física do processo (RSS) é menor com o *Combiner*. Isso ocorre porque o código do *Combiner* é mais enxuto e não necessita dos códigos dos casos de controle. Portanto, o *Combiner* seria mais atrativo para plataformas de hardware limitadas de recursos por não ocupar memória adicional para manter uma base de casos de controle.

Na Tabela 5, são comparados os experimentos entre os algoritmos *Classifier* e *Combiner* (utilizando *Preset swap* + *shift*), em 6 cronogramas gerados pelo FJS. Os cronogramas de produção possuem 34 eventos, referentes ao roteamento de produtos entre as estações do FMC linear e do FMC circular. Um aumento no uso de RSS ocorre devido aos códigos do *Preset* para alguns cronogramas. No entanto, o *Preset* é executado pelo nó *dispatcher*, ou seja, antes das operações do *Combiner*. Da mesma forma, o percentual de utilização do processador (CPU) é semelhante entre os algoritmos. Esse valor normalmente é superior a 100% devido ao processo possuir várias *threads* em núcleos distintos do processador. O *makespan* do algoritmo *Combiner*

(com *Preset swap + shift*) é novamente menor que o do *Classifier* e, portanto, mais eficiente. Cronogramas que encerraram com impasses aparecem na tabela como "*canceled*" (✗). Nestes casos, os valores de *makespan* indicam por quanto tempo os cronogramas puderam ser executados sem impasses (em geral, até 30% do valor de *length*).

Tabela 5 – Comparação entre *Classifier* e *Combiner* (com *Preset swap + shift*).

	Eventos	***Length* (sec.)**	**Agentes**	**CSC**	**CBC**	***Wait***	***Swap***	***Shift***	**RSS (KB)**	**CPU (%)**	**MEM (%)**	***Makespan* (sec.)**
11	34	110	11	✓	-	-	-	-	200.79	153.93	2.60	123.97
	34	110	11	-	✗	-	-	-	199.74	153.70	2.59	40.45
	34	110	11	-	✓	-	✓	✓	200.84	155.16	2.60	122.16
12	34	110	11	✓	-	-	-	-	201.55	152.70	2.61	128.01
	34	110	11	-	✗	-	-	-	201.36	158.80	2.61	40.32
	34	110	11	-	✓	-	✓	✓	200.16	153.00	2.59	125.39
13	34	110	11	✓	-	-	-	-	200.73	152.56	2.60	129.78
	34	110	11	-	✗	-	-	-	200.64	156.50	2.60	41.22
	34	110	11	-	✓	-	✓	✓	201.25	153.73	2.61	127.02
14	34	110	11	✓	-	-	-	-	201.15	155.43	2.60	128.70
	34	110	11	-	✗	-	-	-	200.97	157.70	2.60	39.52
	34	110	11	-	✓	-	✓	✓	200.94	154.33	2.60	125.55
15	34	110	11	✓	-	-	-	-	201.82	156.40	2.61	124.44
	34	110	11	-	✗	-	-	-	199.12	151.70	2.58	38.98
	34	110	11	-	✓	-	✓	✓	200.90	155.43	2.60	123.45
16	34	110	11	✓	-	-	-	-	201.01	154.00	2.60	122.74
	34	110	11	-	✗	-	-	-	201.44	154.60	2.61	38.70
	34	110	11	-	✓	-	✓	✓	201.19	155.43	2.60	125.32

Fonte: Autoria própria.

Outro ponto importante observado na Tabela 5 é que o *makespan* (nos resultados sem impasses) se desvia de *length* devido ao aumento de processamento no sistema (por exemplo, por haverem mais eventos, mais agentes e pela utilização do braço robótico). Isso ocorre porque cada agente no ambiente virtual é lançado como uma *thread* independente, mas vinculada a um único processo comum. No entanto, tais agentes podem ter unidades de *hardware* separadas em um cenário real, assim a carga de processamento poderá ser distribuída entre as unidades.

A Tabela 6 mostra os resultados dos experimentos realizados com o *Combiner* (com *Preset* + *wait*), em 5 cronogramas de produção. Um detalhe é o aumento do número de agentes e, consequentemente, do campo *length* no *Combiner* com *Preset*. Entretanto isso é normal e já era esperado, pois faz parte da estratégia de evitar impasses do método *wait*. Contudo, o *makespan* é equivalente ou inferior ao *length* nos cronogramas avaliados. Os percentuais de MEM e CPU também corroboram com o que já havia sido discutido em resultados anteriores.

Vale destacar que os algoritmos *Classifier* e *Combiner* (sem *Preset*) não puderam executar os cronogramas originais livres de impasses (Tabela 6). Um ponto importante sobre o *Classifier* é que sem um percentual de aceitação (ver ω, na Equação 3), os agentes tendem a selecionar casos de controle por semelhança, que são inadequados para conduzir a produção.

Tabela 6 – Cronogramas completados exclusivamente pelo *Combiner* (com *Preset wait*).

	Eventos	*Length* (sec.)	Agentes	CSC	CBC	*Wait*	*Swap*	*Shift*	RSS (KB)	CPU (%)	MEM (%)	*Makespan* (sec.)
17	10	30	4	✗	-	-	-	-	198.64	150.80	2.57	12.85
	10	30	4	-	✗	-	-	-	198.73	156.60	2.57	13.84
	13	30+10	5	-	✓	✓	-	-	198.24	157.80	2.57	38.67
18	18	50	6	✗	-	-	-	-	200.38	156.70	2.59	12.32
	18	50	6	-	✗	-	-	-	200.48	157.90	2.60	22.98
	21	50+20	7	-	✓	✓	-	-	199.13	155.60	2.58	58.75
19	26	70	8	✗	-	-	-	-	200.12	164.40	2.59	12.90
	26	70	8	-	✗	-	-	-	200.22	165.50	2.59	33.78
	29	70+20	9	-	✓	✓	-	-	198.89	156.60	2.58	81.68
20	26	70	8	✗	-	-	-	-	200.37	155.70	2.59	12.64
	26	70	8	-	✗	-	-	-	198.44	167.50	2.57	33.20
	29	70+20	9	-	✓	✓	-	-	199.47	168.40	2.58	78.00
21	18	50	6	✗	-	-	-	-	200.66	158.80	2.60	12.92
	18	50	6	-	✗	-	-	-	198.88	160.60	2.58	22.67
	21	50+20	7	-	✓	✓	-	-	198.61	158.00	2.57	53.59

Fonte: Autoria própria.

No entanto, os impasses com o *Classifier* ocorrem devido à inexistência de casos de controle na base de casos para os cronogramas experimentados. Assim, o algoritmo *Classifier* só pôde executar em torno de 15% a 33% de cada cronograma até o momento de bloqueio, enquanto que o *Combiner* sem *Preset* executou de 35% a 42%.

A Tabela 7 exibe os resultados para o *Combiner* (sem *Preset*), em cronogramas de produção contendo 30 eventos e *length* de 105 segundos. Os cronogramas executados com o algoritmo *Classifier* foram marcados como "*canceled*", pois a base de casos de controle precisa ser incrementada para que o algoritmo execute sem impasses os cronogramas da tabela.

Tabela 7 – Cronogramas completados exclusivamente pelo *Combiner* (sem *Preset*).

	Eventos	*Length* (sec.)	Agentes	CSC	CBC	*Wait*	*Swap*	*Shift*	RSS (KB)	CPU (%)	MEM (%)	*Makespan* (sec.)
22	30	105	13	✗	-	-	-	-	201.11	148.80	2.60	12.81
	30	105	13	-	✓	-	-	-	200.83	163.57	2.60	111.22
23	30	105	13	✗	-	-	-	-	200.32	159.50	2.59	12.33
	30	105	13	-	✓	-	-	-	201.69	149.96	2.61	112.47
24	30	105	13	✗	-	-	-	-	200.38	152.60	2.59	12.97
	30	105	13	-	✓	-	-	-	201.34	166.73	2.61	113.77
25	30	105	13	✗	-	-	-	-	200.31	163.60	2.59	37.89
	30	105	13	-	✓	-	-	-	201.29	152.36	2.61	110.02
26	30	105	13	✗	-	-	-	-	200.83	152.40	2.60	12.69
	30	105	13	-	✓	-	-	-	200.47	155.96	2.60	114.66

Fonte: Autoria própria.

Neste estudo, os casos de controle são instruções de como os agentes devem processar cada evento do cronograma. Logo, um produto que é processado sequencialmente nas estações $mst1$ e $mst2$, por exemplo, deve possuir um caso de controle escrito exclusivamente para executar essa sequência. Do contrário, o *Classifier* avisará da inexistência de casos na base ou selecionará um caso aproximado que conduzirá a produção enquanto não ocorrer o bloqueio.

Tabela 8 – Outros cronogramas de produção maiores completados pelo *Combiner*.

	Eventos	*Length* (sec.)	Agentes	CSC	CBC	*Wait*	*Swap*	*Shift*	RSS (KB)	CPU (%)	MEM (%)	*Makespan* (sec.)
27	35	105	14	✗	-	-	-	-	202.04	180.30	2.62	53.93
	35	105	14	-	✓	-	-	-	201.24	176.60	2.61	124.37
28	37	105	14	✗	-	-	-	-	200.77	164.60	2.60	45.28
	37	105	14	-	✓	-	-	-	201.67	169.06	2.61	122.16
29	37	105	14	✗	-	-	-	-	201.40	150.90	2.61	15.04
	37	105	14	-	✓	-	-	-	200.23	165.33	2.60	137.03
30	39	115	14	✗	-	-	-	-	201.91	180.60	2.61	15.77
	39	115	14	-	✓	-	-	-	201.43	180.33	2.61	146.87
31	47	125	15	✗	-	-	-	-	202.75	155.80	2.63	14.25
	47	125	15	-	✓	-	✓	-	201.73	183.13	2.61	150.03
32	47	125	15	✗	-	-	-	-	202.75	155.80	2.63	14.25
	47	125	15	-	✓	-	✓	-	202.24	172.93	2.62	158.18

Fonte: Autoria própria.

Por fim, a Tabela 8 exibe os resultados dos experimentos com cronogramas de produção mais longos (entre 35 e 47 eventos). Estes cronogramas somente puderam ser completados sem impasses com o algoritmo *Combiner*. A relação entre os campos *length* e *makespan*, bem como os percentuais de CPU e MEM, e os valores de RSS também são apresentados na tabela. Comparando com as tabelas anteriores, o percentual de CPU e MEM é maior, assim como os valores de RSS. O algoritmo *Combiner* também fechou com valores de *makespan* ligeiramente superiores ao *length*. Essa diferença é devido ao aumento do número de agentes e eventos envolvidos em cada cronograma, o que implica uma maior carga de processamento no sistema.

3.4 AVALIAÇÃO GERAL

O algoritmo *Combiner*, em conjunto com os métodos de *Preset*, se mostrou mais eficiente que o *Classifier* para o conjunto de *benchmarks* experimentados, finalizando todas as instâncias sem a ocorrência de impasses. A Tabela 9 resume os resultados dos experimentos realizados com os 32 cronogramas FJS. Os percentuais médios de RSS, CPU e MEM, para cada solução (*Classifier* ou *Combiner* + *Preset*) estão disponíveis na tabela. Os cronogramas utilizados foram agrupados em seis conjuntos, cada conjunto exibe o desvio percentual médio do *makespan* em relação ao percentual médio de *length*. O percentual de cronogramas concluídos por solução também é exibido para fins de avaliação geral dos resultados. Devido ao emprego dos métodos de *Preset*, foi calculada uma taxa de CPU e memória um pouco maior para a solução *Combiner* + *Preset*. Entretanto, o *Preset* é executado antes do envio dos cronogramas para o sistema multiagente. No caso do *Classifier*, também é importante observar que apenas um caso de controle por cronograma é carregado na memória para que os agentes executem a produção;

cerca de 4,5 a 6,9 kilobytes por caso de controle (não considerados na tabela).

Tabela 9 – Resultados dos experimentos em cronogramas FJS.

		Classifier	***Combiner* + *Preset***
Percentual de uso (%)	**RSS**	2.48%	2.49%
	CPU	153.63%	157.22%
	MEM	2.57%	2.58%
Desvio do complimento ideal por grupo de instâncias (%)	**1–10**	0.99%	-2.03%
	11–16	14.80%	13.47%
	17–21	✗	-13.69%
	22–26	✗	7.07%
	27–30	✗	23.36%
	31–32	✗	23.28%
Instâncias FJS concluídas sem *deadlocks* (%)		50%	100%

Fonte: Autoria própria.

Os grupos de cronogramas marcados com "*canceled*" (✗) não puderam ser executados em sua totalidade pelo algoritmo *Classifier*. Para o grupo 1–10, o desvio do *makespan* foi 0,99% maior no *Classifier*, enquanto que no *Combiner* foi 2,03% menor (mais eficiente). No grupo 11–16, o *Classifier* apresentou um desvio 14,80% maior do *length*, enquanto que o do *Combiner* + *Preset* foi de 13,47% para os mesmos cronogramas (ou seja, 9% menor que *Classifier*). No grupo 17–21, apenas o *Combiner* conseguiu concluir os cronogramas, levando 13,69% menos tempo que o estimado pelo FJS. Nos grupos 22–26 ($>$ 7,07%), 27–30 ($>$ 23,36%) e 31–32 ($>$ 23,28%), o *Combiner* também executou com sucesso os cronogramas livres de impasses, apesar de apresentar percentuais superiores ao *length* médio. Os resultados mostraram que a solução *Combiner* + *Preset* executou 100% dos cronogramas utilizados neste estudo, sem impasses. Em comparação, o *Classifier* executou 50% de todos os cronogramas devido à falta de casos de controle livres de impasses na base de casos. Porém, apesar de ser trabalhoso desenvolver novos casos de controle, o algoritmo *Classifier* tem o potencial para resolver outras situações envolvendo tomadas de decisão no MAS.

Tabela 10 – Complexidade de tempo dos algoritmos propostos.

		Melhor caso	**Pior caso**
Preset	***Swap***	$\Omega(4n + 14)$	$O(n^2 + 4n + 21)$
	Shift	$\Omega(4n + 12)$	$O(n^2 + 8n + 29)$
	Wait	$\Omega(10n + 69)$	$O(10n + 86)$
Classifier		$\Omega(n^2 + 5n + 48)$	$O(n^2 + 7n + 110)$
Combiner		$\Omega(2n + 11)$	$O(3n + 90)$

Fonte: Autoria própria.

Por fim, a Tabela 10 apresenta dados sobre a complexidade temporal dos algoritmos propostos. As notações Ω e O foram utilizadas para expressar o limite inferior (melhor caso) e limite superior (pior caso) dos algoritmos. A complexidade do *Classifier* e do *Combiner* referem-se a um único agente MAS. No caso do *Classifier*, os valores incluem a complexidade de tempo do caso de uso mais extenso da base. O cálculo do *Preset* aparece separado do *Combiner* porque os ajustes ocorrem antes do envio do cronograma ao MAS. As operações da tabela de decisão usada pelo *Combiner* para roteamento de produtos, instruções condicionais, atribuições e operações simples foram computadas como tempo constante. As operações mais complexas foram contabilizadas de acordo com a complexidade de tempo definida em documentos sobre o CPython atual e operações específicas do FMS virtual não foram consideradas no cálculo.

3.5 CONCLUSÕES DO CAPÍTULO

Este capítulo apresentou um estudo preliminar sobre sistemas multiagentes, onde investigou-se a sua aplicabilidade em problemas de escalonamento e de controle da manufatura. Para isso, um MAS distribuído foi desenvolvido para gerenciar a produção de um FMS virtual, de acordo com cronogramas gerados por um algoritmo de escalonamento preditivo clássico. Os recursos utilizados para o roteamento de produtos no FMS também foram incluídos no escalonamento e um controle de consenso sem líderes garantiu a cooperação entre agentes na execução das operações de produção.

Dois algoritmos para evitar impasses no FMS foram propostos e comparados. O primeiro algoritmo, chamado *Classifier*, utilizou a teoria de Dempster-Shafer modificada com um valor de aceitabilidade para a função de crença. Os agentes foram capazes de converter multiparâmetros em resultados de decisão que indicavam casos de controle livres de impasses para lidar com a produção. O segundo algoritmo, chamado *Combiner*, utilizou uma tabela de decisão para o roteamento de produtos no FMS e um mecanismo de controle que operou com cronogramas de produção modificados, evitando potenciais impasses no sistema.

Para avaliar a eficiência dos algoritmos, foram realizados experimentos com 32 cronogramas de produção. Os cronogramas possuíam de 10 a 47 eventos e os resultados dos experimentos mostraram que o *Combiner* + *Preset* foi mais eficiente para o conjunto de *benchmarks* testados, pois possibilitou a completa execução de todos os cronogramas de produção. Entretanto, apesar da falta de casos de controle ter limitado o desempenho do *Classifier*, não significa que a teoria de Dempster-Shafer é inadequada para tomadas de decisões no MAS.

Este estudo mostrou que MAS são apropriados para a descentralização de sistemas de controle complexos na manufatura, especialmente em ambientes com equipamentos fisicamente distribuídos e com requisitos de flexibilidade. Ficou comprovado que, com o suporte de um controle de consenso adequado, um comportamento global coletivo pode ser alcançado a partir do comportamento individual dos agentes. Por fim, as abordagens apresentadas neste estudo contemplaram algumas necessidades identificadas para o FMS virtual, como a inclusão dos recursos de transporte no escalonamento, flexibilidade na produção e o controle de certos tipos de impasses. Entretanto, ainda é necessário aperfeiçoamentos para garantir maior abrangência das técnicas em novos cenários de produção.

4 PROGRAMAÇÃO FLEXÍVEL E REAJUSTAMENTOS

Sistemas multiagentes são adequados para a descentralização de sistemas flexíveis de manufatura, pois foi comprovado que a meta global dos agentes, que é o controle da produção, é alcançada a partir de comportamentos individuais definidos durante o processo de escalonamento. Entretanto, algumas melhorias relacionadas à arquitetura multiagente e ao controle de impasses devem ser implementadas para garantir a replicabilidade das abordagens em outros ambientes de produção. Este estudo propõe um MAS distribuído, de arquitetura em camadas, para um FMS com produção controlada por ordem. Três filtros de reajustamento de cronogramas preditivos de produção foram desenvolvidos para garantir a flexibilidade e reduzir o risco de impasses no sistema. Experimentos foram conduzidos em uma plataforma de experimentação de FMS para avaliar a abordagem proposta. O capítulo encerra com a discussão dos resultados e as conclusões.

4.1 CARACTERÍSTICAS DO TRABALHO

Muitas abordagens propostas para tratar problemas de escalonamento e de controle da manufatura são centralizadas, pois consideram a existência de uma única entidade decisória atuando no controle do ambiente de produção. Com a descentralização dos sistemas de controle da manufatura e o surgimento de várias entidades de decisão distribuídas, estas abordagens centralizadas se tornaram inadequadas para FMSs; devido aos requisitos de flexibilidade, ambientes de produção dinâmicos e a necessidade de tratar conhecimento distribuído. Por esse motivo, muitas pesquisas com foco em reescalonamento (em inglês, *rescheduling*) buscaram projetar ou adaptar métodos heurísticos para processar informações no FMS em tempo de execução e criar novos cronogramas para resolver impasses no sistema. No entanto, embora algumas pesquisas tenham obtido resultados eficientes, como em (DURASEVIć; JAKOBOVIć, 2020; LI *et al.*, 2021; UHLMANN; FRAZZON, 2018; SUN *et al.*, 2021) e (MESSINIS; VOSNIAKOS, 2020), as heurísticas propostas geralmente se limitam a ambientes de produção específicos, e são difíceis de adaptação a outros cenários, ou de comparação com outras abordagens.

Este estudo está relacionado com o terceiro e quarto artigos da lista de trabalhos desenvolvidos (Seção 1.3). Na literatura há um campo de pesquisa que visa o controle da produção e a recuperação de impasses, definido como "reescalonamento" (em inglês, "*rescheduling*"), com métodos para obter novos cronogramas de produção em tempo de execução e recuperar o

sistema de impasses. Entretanto, no presente estudo define-se um segundo campo de pesquisa voltado ao "reajustamento" (em inglês, "*readjustment*"), com métodos para ajustar cronogramas preditivos e evitar que possíveis impasses e desvios ocorram no sistema, sem a necessidade de reescalonamento. A Tabela 11 apresenta as características de cada campo de pesquisa.

Tabela 11 – Características do reescalonamento e do reajustamento.

	Reescalonamento	**Reajustamento**
Abordagem	Distribuída *	Semi-distribuída
Informação	Parcial	Completa
Escalonamento	Runtime	Preditivo
Adaptação	Difícil	Viável

* Em algumas áreas pode ser tratado de modo centralizado

Fonte: Autoria própria.

A ausência de literatura sobre reajustamento de cronogramas preditivos para FMSs sugere que abordagens com esse enfoque ainda não foram exploradas, ou ainda não obtiveram o reconhecimento merecido. No entanto, é desejável que determinados problemas de escalonamento possam ser resolvidos diretamente nos cronogramas de produção. Pois algumas questões podem ser melhor tratadas antecipadamente, na fase de planejamento, para não prejudicar a capacidade de previsibilidade e flexibilidade do sistema de produção.

É possível caracterizar problemas de reajustamento como semidistribuídos, ao combinar escalonamento preditivo com IA distribuída. Algoritmos de escalonamento preditivo são centralizados, o que significa que dependem de informações completas do FMS para a obtenção dos cronogramas. Apesar de garantirem valores ótimos de *makespan* (ou muito próximos de ótimos), a execução completa dos cronogramas depende de condições ideais de operação no ambiente produtivo. No entanto, restrições, interrupções, atrasos e outros tipos de impasses podem causar desvios das metas inicialmente definidas para o escalonamento. Nesse caso, estratégias de reajustamento e abordagens da IA distribuída (por exemplo, MAS) podem garantir as adaptações e condições necessárias para a execução de cronogramas reajustados.

Como o funcionamento da IA distribuída, incluindo os sistemas multiagentes, depende de um determinado conjunto de módulos (ou partes) para resolver de forma cooperativa os problemas de produção, abordagens inteligentes para problemas de escalonamento e controle da produção podem ser implementadas como módulos adicionais do MAS. Entretanto, é necessário que a arquitetura MAS seja flexível o suficiente para permitir esta separação. Caso contrário, é

provável que as soluções continuem restritas a ambientes específicos e difíceis de comparação ou de aprimoramento por outros pesquisadores.

4.1.1 Arquitetura MAS

Uma dificuldade significativa ao projetar controles para FMSs é a alta complexidade que esses sistemas apresentam, e isso pode ser um entrave para a descentralização com multiagentes. Entretanto, uma arquitetura MAS em camadas pode facilitar a adaptação do sistema e a inclusão de abordagens inteligentes de escalonamento e controle. Portanto, este estudo também propõe uma arquitetura MAS de três camadas para separar o escalonamento e o controle MAS dos métodos de reajustamento dos cronogramas de produção (ou de reescalonamento, se for o caso). As camadas da arquitetura MAS proposta são descritas a seguir:

1. Camada de comunicação: implementa quatro tópicos de comunicação para a troca de mensagens entre agentes. Os tópicos "*synchrony*", "*signaling*" e "*notification*" são empregados no controle de consenso do MAS. O tópico *notification* também é utilizado para a troca de mensagens entre agentes (por exemplo, para notificar sobre a chegada de uma peça). O tópico "*broadcast*" é utilizado para recepção dos cronogramas de produção. A camada de comunicação se baseia na arquitetura dos agentes apresentada no Capítulo 3.

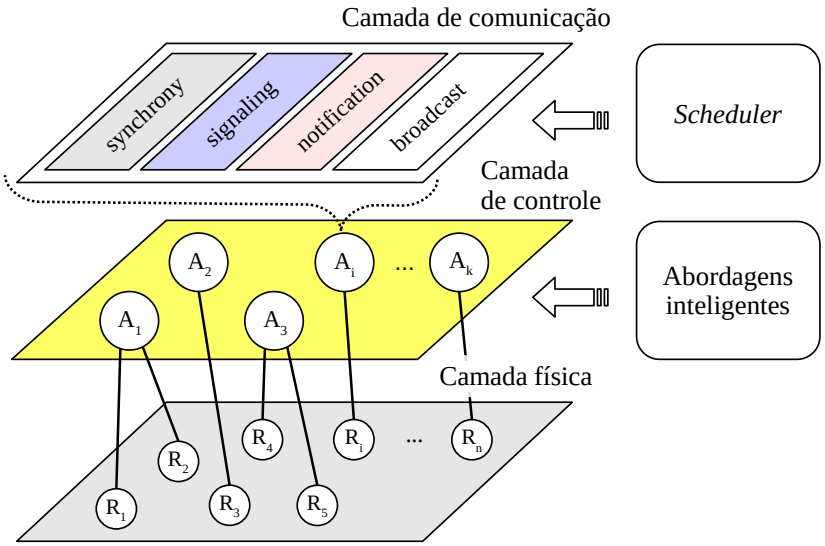
2. Camada de controle: associa cada agente a um ou mais recursos de fabricação do FMS, que podem ser estações de fabricação, máquinas-ferramentas, dispositivos de identificação (leitor NFC ou detector de cores) ou locais específicos na planta (por exemplo, um local de entrega de peças). A camada também é responsável por executar as funcionalidades e decisões tomadas pelos agentes para controlar os recursos de manufatura. Por exemplo, ejetar um produto da esteira ou acionar um forno industrial.

3. Camada física: Fornece abstração de hardware para a camada de controle, para operar os recursos de manufatura através de comandos de alto nível executados pelos agentes.

A Figura 10 apresenta uma visão geral da arquitetura MAS, tal que $A = \{A_1, A_2, A_3, ..., A_i, ..., A_k\}$ identifica os agentes na camada de controle. A camada de comunicação é idêntica em cada agente A_1, exceto por alguns tipos de mensagens em *notification*, específicas do agente e recursos associados. A camada física, na parte inferior da figura, repre-

senta os recursos de fabricação ($R_1, R_2, R_3, ..., R_i, ..., R_n$). Cada agente está associado a pelo menos um recurso de fabricação e pode operá-lo de acordo com as necessidades da produção.

Figura 10 – Visão geral da arquitetura MAS de três camadas.



Fonte: Sousa e Oliveira (2023).

A camada de comunicação fornece serviços para a camada de controle, que utiliza serviços da camada física. O *framework* ROS foi utilizado neste estudo, mas outros métodos de comunicação podem ser adaptados para troca de mensagens via tópicos, compatível com o modelo *publish-subscribe* (por exemplo, *Message Queuing Telemetry Transport* - MQTT). O bloco denominado "*smart approaches*", associado à camada de controle, representa os processos cognitivos individuais ou coletivos e funcionalidades dos agentes, enquanto que o bloco "*scheduler*", associado à camada de comunicação, trata do planejamento da produção no FMS. No presente estudo, o escalonamento da produção é realizado por um nó chamado "*dispatcher*", que reajusta os cronogramas de produção quando impasses forem identificados.

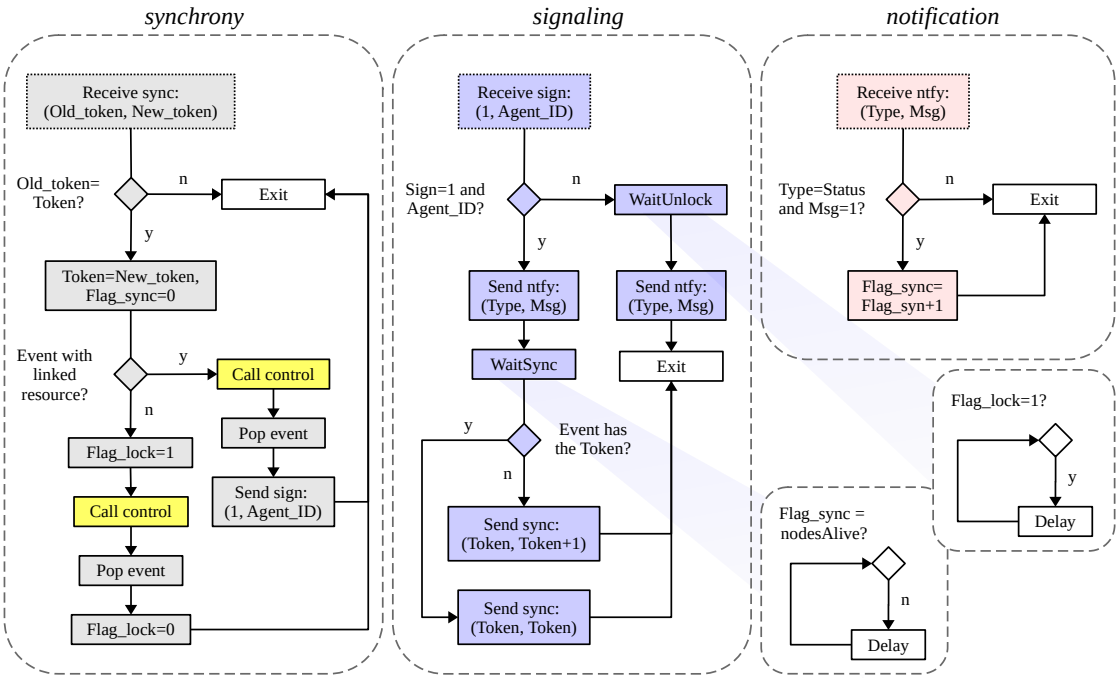
4.1.2 Modelo de consenso

O controle de consenso da presente pesquisa pode ser classificado como "consenso em tempo finito pré-definido", de acordo com as topologias de consenso para o MAS levantadas por (AMIRKHANI; BARSHOOI, 2022; LI; TAN, 2019). No consenso de tempo finito, os agentes concordam com um estado consistente em um intervalo de tempo finito. No entanto, definir parâmetros relacionados com o tempo de convergência do MAS para o protocolo de consenso é um desafio (AMIRKHANI; BARSHOOI, 2022). Segundo Li e Tan (2019), o projeto e a análise

de algoritmos de consenso de tempo finito são mais complexos do que os de consenso assintótico, cujo valor da taxa de convergência é independente do tempo. No entanto, para sistemas de manufatura, o tempo de convergência do MAS deve ser mais rigoroso e estar dentro de um tempo finito para garantir a previsibilidade na produção. Na convergência em tempo finito pré-definido, um caso específico de consenso em tempo finito, o tempo exato para obter consenso é um parâmetro a priori do protocolo de consenso (AMIRKHANI; BARSHOOI, 2022).

O controle de consenso deste estudo é baseado em eventos processados em períodos de tempo pré-definidos nos cronogramas de produção gerados e na troca de mensagens entre agentes no MAS. O processamento de um evento é iniciado quando os agentes atualizam suas entradas de controle ao receberem um sinal de sincronização. De acordo com Dorri *et al.* (2018), a sincronização significa que as ações de cada agente estão alinhadas no tempo com as dos outros agentes. No controle de consenso proposto, a sincronização garante que todos os agentes convirjam para um valor em comum de *token* (indexador dos eventos), caracterizando assim um consenso de tipo completo. Os tópicos *synchrony*, *signaling* e *notification* da camada de comunicação são empregados no consenso, todos os agentes podem ler e escrever nesses tópicos. A Figura 11 apresenta as operações do controle de consenso.

Figura 11 – Modelo de controle de consenso.



Fonte: Sousa e Oliveira (2023).

Quando o bloco de código associado ao tópico *synchrony* recebe uma mensagem de

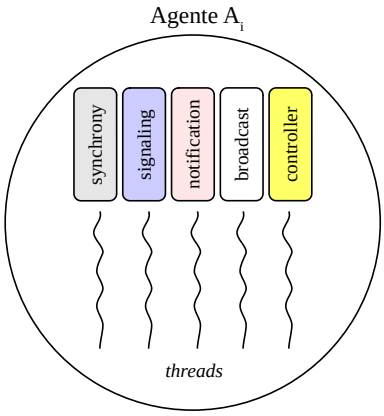
sincronização, os agentes verificam se o *token* antigo contido da mensagem é igual ao *token* atual. Nesse caso, os agentes atualizam o *token* atual com o novo valor relatado na mensagem de sincronização. Caso contrário, o *token* já está atualizado e os agentes nesta condição são liberados de processar o evento atual até que uma nova mensagem de sincronização seja recebida. Depois que o *token* é atualizado, uma *flag* de sincronização é reinicializada e, em seguida, os agentes verificam se a identificação de agente contida no evento atual corresponde ao seu ID de agente. Nesse caso, o agente identificado no evento executa a operação de produção invocando os recursos necessários. O agente então remove o evento atual do cronograma e sinaliza a conclusão da tarefa. Porém, os demais agentes também verificam de algum procedimento de controle é necessário para o evento corrente (por exemplo, preparar determinado recurso para receber o produto). Por fim, antes de excluir o evento do cronograma, os agentes ativam uma *flag* de bloqueio que impede que o *token* seja atualizado até a conclusão da operação de produção atual.

Quando uma mensagem de conclusão de tarefa é recebida no bloco de código do tópico *signaling*, os agentes verificam se ela contém seu ID de agente. Caso contrário, o agente aguarda a desativação da *flag* de bloqueio (em *WaitUnlock*). Posteriormente, o agente envia uma mensagem de *status* que incrementa o valor da *flag* de sincronização (no bloco *notification*). Esta *flag* de sincronização indica o número de agentes ativos que verificaram o último evento. Porém, quando a mensagem de conclusão da tarefa contém o ID do agente, ele também participa do incremento da *flag* de sincronização e aguarda o sincronismo dos demais agentes (em *WaitSync*). Caso o evento atual tenha o mesmo valor de *token* do evento anterior, o agente envia uma mensagem para o tópico *synchrony* informando o *token* atual e o novo (*token*, *token*+1). Caso contrário, o agente enviará uma mensagem de sincronização repetindo os valores (*token*, *token*). Cabe destacar que o evento atual não é mais o mesmo usado do bloco *synchrony*, porque lá ele foi excluído do cronograma. Portanto, se o evento atual (no *signaling*) tiver o mesmo valor de *token*, trata-se de um evento paralelo para outro agente do MAS. Este mecanismo é necessário porque o *dispatcher* envia o cronograma completo para os agentes. Pois, determinados eventos ainda podem exigir a ação de agentes específicos (por exemplo, para notificar que o produto foi recebido).

Por fim, no bloco do tópico *notification*, quando um agente recebe uma mensagem de notificação do tipo "*status*" contendo o valor 1 (que significa "on"), ele incrementa sua *flag* de sincronização. Outras mensagens são utilizadas no bloco *notification* para a comunicação entre agentes, mas são mensagens que não tem relação com o consenso. Sempre que um novo cronograma de produção é iniciado, ou repetido após ter sido concluído, o *token* é zerado e o

processo de consenso e execução da produção é repetido. Um aspecto essencial da arquitetura é que o modelo de agente executa os blocos *synchrony*, *signaling*, *notification*, *broadcast* e *control* em *threads* independentes (Figura 12). Portanto, o processamento sequencial não atrasa a execução de cronogramas em eventos individuais ou paralelos, a menos que impasses ou desvios ocorram por falhas nos recursos de fabricação da camada física.

Figura 12 – Modelo de agente.



Fonte: Autoria própria.

Um modelo de agente bem estruturado e alinhado ao modelo de consenso garante escalabilidade. Mais detalhes sobre a arquitetura dos agentes estão disponíveis no Apêndice A.

O FMS deve ser suficientemente escalável para permitir mudanças como, por exemplo, a adição de novos FMCs. A escalabilidade é uma característica importante que deve ser considerada na fase de projeto, pois é difícil adicioná-la posteriormente. Sem um MAS, alcançar um equilíbrio no FMS com esquemas de controle convencionais é um desafio. A capacidade de convergência do MAS em períodos de tempo pré-definidos também é essencial para FMSs que dependem de pontos de sincronismo para evoluir seus processos de produção. Do contrário, não há como garantir o sequenciamento das operações e um atendimento preciso das demandas quanto ao tempo de produção.

4.1.3 Plataforma de experimentação de FMS

Este estudo utiliza uma plataforma física de experimentação de fábrica da Fischertechnik, adquirida após os estudos relacionados com o FMS virtual. Originalmente a plataforma possuía topologia centralizada, com controladores que se comunicavam exclusivamente com um controlador central. O ciclo de produção da plataforma era fixo, ou seja, não havia flexibilidade para mudar a sequência de produção. Para desenvolver novos controles e funcionalidades

avançadas de interesse da pesquisa era preciso conhecimento avançado em C++. Entretanto, o processo de compilação de códigos C++ utilizando um *cross-compiler* para plataforma ARM é extremamente lento. Os códigos também não podem ser compilados diretamente nos controladores porque, apesar de rodarem Linux, possuem *firmware* proprietário e são limitados em memória e processamento.

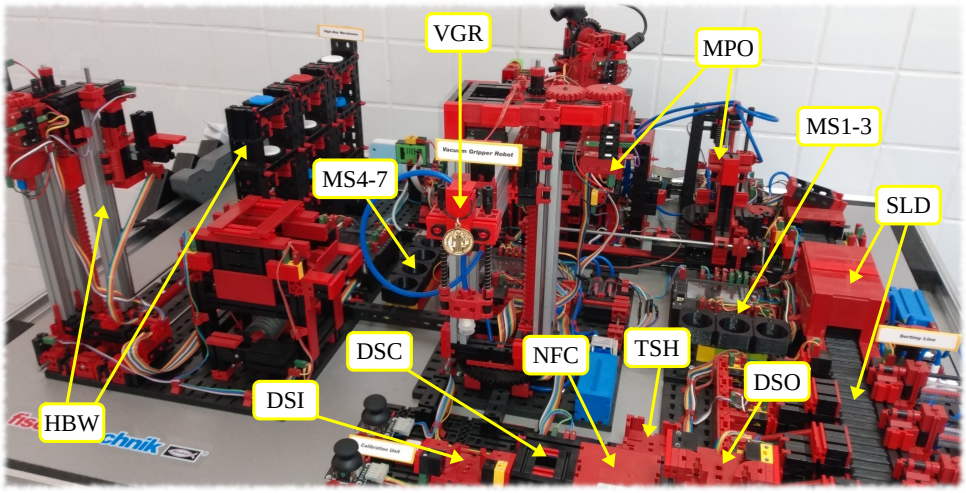
Neste estudo a plataforma de experimentação foi modificada para operar como um FMS com produção controlada por ordem (mais detalhes no Apêndice B). Um módulo Python, chamado *ftrobopy4*, de uso geral para controladores TXT 4.0 da Fischertechnik, foi incorporado ao sistema dos controladores e utilizado para fazer a ponte com o *firmware* proprietário, possibilitando o rápido desenvolvimento de aplicações e a integração ao MAS. As características da nova plataforma de experimentação de FMS são descritas a seguir:

- *Vacuum gripper robot* (VGR): estação que possui um robô com garra de sucção a vácuo para a movimentação de peças no FMS.
- *High-bay warehouse* (HBW): estação composta por um armazém automatizado de estantes altas, com 3 x 3 *slots*, para armazenamento de peças em contêineres.
- *Multi-processing station* (MPO): estação de multiprocessamento que simula um forno industrial, mesa giratória com atuador e bancada de usinagem.
- *Sorting line and detection* (SLD): consiste em uma linha de classificação com esteira, câmara de detecção de cores das peças e três pistões pneumáticos.
- General-purpose manufacturing stations (MS1-MS7): sete estações de manufatura com LEDs para indicar usinagem e compartimentos que acomodam peças.

A fábrica também possui um sensor de detecção de cores (DCS), leitor/gravador de *tags* (NFC), ponto de descarte de peças (TSH), ponto de chegada de peças (DSI) e ponto de coleta de peças (DSO). No total, 14 pontos de processamento de peças (DSI, DSC, NFC, HBW, MPO, SLD, MS1, MS2, MS3, MS4, MS5, MS6, MS7, DSO) estão envolvidos no planejamento da produção. A Figura 13 mostra as localizações físicas dos recursos no FMS.

Pequenas peças cilíndricas (altura = 13,9 mm, diâmetro = 26,1 mm) de cores variadas (branco, vermelho e azul) são movidas no FMS para simular a produção. Cada peça possui uma etiqueta de identificação NFC individual que registra a cor e o histórico de produção, com informações de data e hora.

Figura 13 – Plataforma de experimentação para o FMS.



Fonte: Autoria própria.

As características físicas e funcionais dos recursos de fabricação disponíveis na plataforma, permitem uma classificação de acordo com o tipo de *buffer*:

- *Buffer* limitado: retém a peça por certo tempo de acordo com a capacidade do recurso (por exemplo, contêineres no armazém HBW).
- *Buffer* transitório limitado: retém a peça somente até o próximo recurso ficar disponível (por exemplo, uma estação de fabricação).
- *Buffer* nulo: recurso de fabricação sem *buffer* que ocupa o sistema de transporte durante a operação (por exemplo, sensor de detecção de cores).

De acordo com essa classificação, os recursos DSO, TSH e HBW são do tipo *buffer* limitado com capacidade igual a 1 (uma peça). Os recursos MS1, MS2, MS3, MS4, MS5, MS6 e MS7 são do tipo *buffer* transitório limitado com capacidade igual a 1, pois podem reter uma peça recém-usinada por um determinado período de tempo, desde que o recurso permaneça ocioso durante o próximo incremento de *token*. Por fim, os recursos DSI, DSC, MPO, SLD e NFC são do tipo *buffer* nulo, pois dependem do robô de VGR para suas operações de produção. Neste caso, o robô fica indisponível para outras operações paralelas até o término da operação atual.

No FMS, os recursos de fabricação são comandados pelos agentes para executarem as tarefas definidas nos cronogramas. O MAS possui 11 agentes, identificados de A_1 a A_{11}. O agente A_1 é responsável pelo sistema de transporte e coordena o robô de VGR para mover as peças para os demais recursos do FMS. DSI, DSC, NFC, DSO e TSH também estão associados

a A_1 porque o controlador de VGR está fisicamente conectado a eles na fábrica. O agente A_2 coordena os recursos da estação MPO, incluindo um atuador pneumático que empurra a peça para o transportador em SLD. O agente A_3 coordena o robô de HBW na movimentação de peças das estantes para a esteira e vice-versa. O agente A_4 coordena os recursos da estação SLD para classificar as peças por cor. Além disso, as estações $MS1$–$MS7$ são coordenadas pelos agentes A_5–A_{11}. Essas estações foram impressas em 3D e adicionadas ao FMS para aumentar o seu potencial como plataformas de experimentação.

Na prática, os agentes das estações VGR, MPO, HBW e SLD rodam nos controladores associados a elas. As estações $MS1$–$MS7$ não tem controladores, os agentes rodam em um PC e comunicam com as estações via NodeMCU (placa com suporte a Wi-Fi). Um roteador Wi-Fi ou computador configurado como ponto de acesso fornece a rede para o FMS. A produção é coordenada de acordo com os cronogramas definidos pelo *dispatcher* (ver subseção 3.1.3).

4.2 FILTROS DE REAJUSTAMENTO

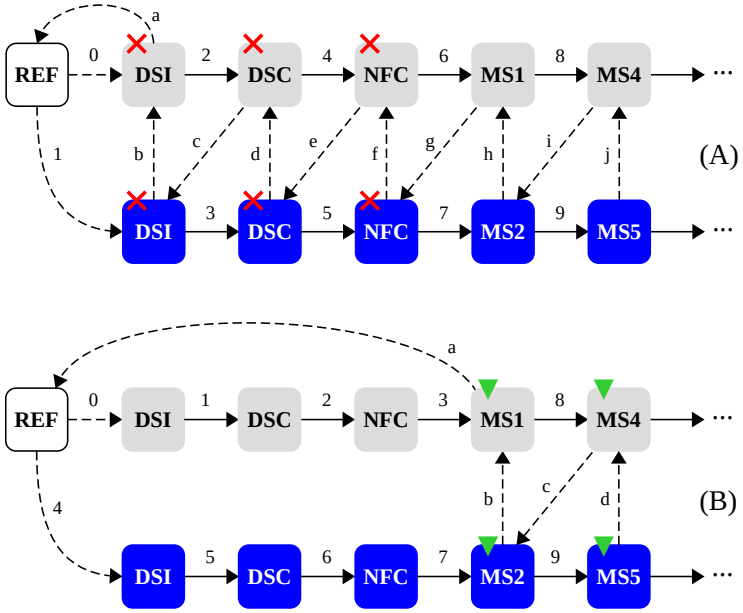
Esta seção apresenta três tipos de filtros de reajustamento aplicados em cronogramas de produção preditivos para reduzir impasses em FMSs. Os filtros operam conjuntamente, cada qual com suas funções, com o objetivo de que os cronogramas gerados no escalonamento sejam executados integralmente, livres de bloqueios. A abordagem focada no reajustamento de cronogramas preditivos é uma alternativa às soluções de reescalonamento computacionalmente caras, que geralmente são limitadas a ambientes de produção específicos, e difíceis de serem reproduzidas em outros ambientes.

4.2.1 Filtro de serialização

O primeiro filtro para a produção livre de impasses, chamado de "filtro de serialização", identifica se os cronogramas de produção contêm eventos associados a recursos de *buffer* nulo. Nesse caso, o método serializa esses eventos para que não ocorram paralelamente a outros eventos do cronograma. Caso contrário, o robô de VGR responsável pelo transporte de peças no FMS não poderá movimentar novas peças para cumprir outras operações agendadas porque estará ocupado com a peça atual. A Figura 14 ilustra o funcionamento do filtro de serialização para um cronograma de produção com dois tipos de peças.

A primeira peça é fabricada de acordo com a sequência $REF \prec DSI \prec DSC \prec$

Figura 14 – Funcionamento do filtro de serialização.



Fonte: Autoria própria.

$NFC \prec MS1 \prec MS4 \prec DSO \prec REF$ (na cor cinza), enquanto que a segunda peça é fabricada de acordo com a sequência $REF \prec DSI \prec DSC \prec NFC \prec MS2 \prec MS5 \prec DSO \prec REF$ (em azul). As setas numeradas indicam o movimento do robô de VGR carregado (com um peça), enquanto as setas tracejadas representam o movimento do robô descarregado (sem peça). Dado um cronograma inicial de produção (Figura 14-A), o robô deverá mover-se do ponto de referência (REF) até DSI para selecionar o primeiro tipo de peça (transição 0). A passagem de REF para DSI corresponde ao primeiro evento do cronograma. Posteriormente, o robô deverá retornar vazio para REF (transição "a"), pegando o segundo tipo de peça em DSI (transição 1). Porém, o robô carregado com a primeira peça retirada de DSI não poderá descarregar para pegar a segunda peça, o que causará um *deadlock* no FMS. Da mesma forma, o problema se repete em DSC e NFC devido ao *buffer* nulo dos recursos e ao compartilhamento do robô de VGR. Os algoritmos de escalonamento preditivo geralmente não são projetados para identificar e prevenir esse problema.

O filtro de serialização garante que uma peça seja processada sequencialmente em recursos de *buffer* nulo, até que seja depositada em um recurso de *buffer* limitado ou transitório limitado (Figura 14-B). Para o primeiro tipo de peça, o robô de VGR realizará sequencialmente as transições de 0 a 3 até depositar a peça na estação MS1. MS1 e as outras estações são classificadas como *buffer* transitório limitado. Em seguida, o robô de VGR deverá se mover descarregado para REF (transição "a"), pegar uma peça do segundo tipo (transição 4) e movê-la sequencialmente

pelos recursos de *buffer* nulo até descarregar a peça na estação MS2. Logo após, o robô de VGR recuperará a peça que estava em MS1 (transição "b") e a depositará em MS4 (transição 8). Em seguida, retornará descarregado para pegar o outro tipo de peça deixado em MS2 (transição "c"), para depositar em MS5 (transição 9), continuando até o término do cronograma.

Algorithm 5 *Serializer filter*

```
1: E ← [E_0, ..., E_i, ..., E_f]                          ▷ cronograma de produção E
2: B ← [0, 1, 2, 3, 4, 5, 6]                              ▷ recursos de buffer nulo
3: function EXEC_SERIALIZER(L)                            ▷ executa a serialização de eventos no cronograma
4:     for (j ← 0,|J|) do
5:         for (g ← 0,|L_j|) do                           ▷ obtém grupo de eventos do produto
6:             idx ← E.index(L_jg0)
7:             for (e ← 0,|L_jg|) do
8:                 E.remove(L_jge)                        ▷ remove de E os eventos do grupo atual
9:             end for
10:            for (e ← |L_jg|,0) do
11:                E.insert(idx, L_jge)                   ▷ insere o grupo em E a partir do índice idx
12:            end for
13:        end for
14:    end for
15:    return update_schedule()
16: end function
17: function CHECK_METHOD()                               ▷ verifica se há bloqueios por buffer nulo
18:    L,aux ← [ ]
19:    for (j ← 0,|J|) do
20:        gpo ← [ ]
21:        for (i ← 0,|E|) do
22:            if (E_{i[job]} = j) and (E_{i[agent]} ∈ B) then   ▷ verifica se o agente do evento corrente está em B
23:                gpo ← append(E_i)
24:                for (k ← i + 1, |E|) do                ▷ varre E a partir do índice i + 1
25:                    if (E_{k[job]} = j) and (E_{k[agent]} ∋ B) then   ▷ se agente não está em B, é buffer limitado
26:                        if (|gpo| = 1) and (gpo0_{[agent]} ∈ B) then
27:                            idx ← E.index(gpo0_0)      ▷ se é único em gpo, ligar com o evento anterior
28:                            for (m ← idx − 1, 0) do    ▷ busca evento anterior de mesmo job
29:                                if (gpo0_{[job]} = E_{m[job]}) then
30:                                    gpo.insert(0, E_m) ▷ insere como primeiro no grupo
31:                                    break
32:                                end if
33:                            end for
34:                        end if
35:                        aux ← append(gpo)
36:                        gpo ← [ ]
37:                    end if
38:                    break
39:                end for
40:            end if
41:        end for
42:        if (|gpo| > 0) then
43:            aux ← append(gpo)
44:        end if
45:        L ← append(aux)
46:        aux ← [ ]
47:    end for
48:    if (|L| > 0) then                                  ▷ se é maior que zero, foram encontrados bloqueios
49:        return exec_serializer(L)                      ▷ chama a função de serialização
50:    else
51:        return E
52:    end if
53: end function
```

O filtro de serialização é apresentado no Algoritmo 5. Dado um cronograma de produção

E, e uma lista B de agentes associados a recursos de *buffer* nulo, o filtro de serialização executará três funções para reajustar o cronograma. Primeiro, o filtro utiliza a função $check\_method()$ para verificar se há recursos de *buffer* nulo no cronograma. A função verifica os eventos definidos para cada peça ou produto j, até encontrar um evento em que o ID do agente ($E_{i_{[agent]}}$) esteja contido em B (linhas 18–21). Se encontrado, a função inclui o evento no grupo gpo (linha 22) e busca outros eventos do mesmo tipo relacionados a j. A busca continua até que a função encontre um evento j que não tenha o ID de agente associado a B, significando que foi identificado um evento associado a um recurso de *buffer* limitado (linhas 23 e 24). Em seguida, a função verifica se o evento de *buffer* nulo incluído em gpo é único; ou seja, não há outros para completar o grupo (pelo menos um par). Neste caso, o evento de gpo é vinculado sequencialmente a um evento anterior do mesmo job (produto), de forma que o evento de *buffer* nulo não fique isolado no cronograma. Assim, o evento anterior é posicionado no início do grupo, seguido do evento de *buffer* nulo (linhas 25 a 33). Assim, se existirem recursos de *buffer* nulo no cronograma em análise, eles são separados em dois ou mais grupos. A função monta uma lista L com os respectivos eventos que devem ser processados sequencialmente para cada produto (linhas 35 e 44). Neste caso, o reajustamento do cronograma é necessário se L contiver eventos (linha 48).

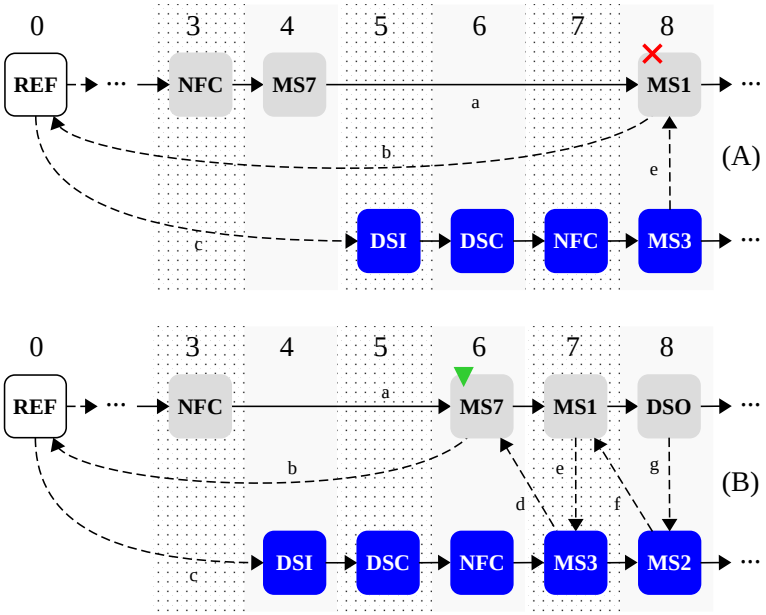
Após, o filtro de serialização utiliza a função $exec\_serializer()$ para ajustar o cronograma de produção. Para cada peça ou produto j, a função obtém o índice de posição (idx) do primeiro evento (L_{jg0}) de cada grupo salvo em L (linhas 3 a 5). A função então retirará todos os eventos relacionados a j do escalonamento inicial (linhas 6 e 7) e os inserirá novamente em ordem inversa, do último para o primeiro, nos índices de posição obtidos anteriormente (linhas 9 e 10). Por fim, o filtro de serialização atualizará os tempos de início ($E_{i_{[start]}}$) e os *tokens* ($E_{i_{[token]}}$) dos eventos no cronograma modificado.

4.2.2 Filtro de podagem

O segundo método para produção livre de impasses no FMS estudado é chamado de “filtro de podagem”. Este método busca identificar eventos com recursos de *buffer* nulo organizados contiguamente a eventos de *buffer* limitado ou transitório limitado relacionados com o mesmo produto ($E_{i_{[job]}}$). Nesse caso, o filtro quebrará a contiguidade dos eventos reajustando o cronograma, de forma que o evento de *buffer* nulo seja priorizado e organizado em paralelo com eventos de outros produtos. Em outras palavras, o método “poda” o último evento de *buffer* limitado contíguo ao evento de *buffer* nulo do mesmo produto ou peça. O evento removido obtém

outro número de *token* e é reposicionado longe do evento de *buffer* nulo. A Figura 15 ilustra o funcionamento do filtro de podagem para um cronograma de produção com dois tipos de peças.

Figura 15 – Funcionamento do filtro de podagem.



Fonte: Autoria própria.

Elas são fabricadas com as sequências $REF \prec DSI \prec DSC \prec NFC \prec MS7 \prec MS1 \prec DSO \prec REF$, para o primeiro tipo de peça, e $REF \prec DSI \prec DSC \prec NFC \prec MS3 \prec MS2 \prec DSO \prec REF$, para o segundo tipo. Ao executar o cronograma de produção sem reajustes (Figura 15-A), o robô de VGR deverá sair do ponto de referência (REF), pegar uma peça e movimentá-la até chegar em NFC. Com o incremento do *token* para três, o robô de VGR moverá a peça de NFC para MS7. Em seguida, com o incremento do *token* para quatro, a peça poderá ser processada em MS7. O robô de VGR então descarregará a peça na estação MS1 (transição "a"), que é o próximo recurso da sequência de fabricação do primeiro tipo de peça. Então, o robô de VGR retornará descarregado para REF (transição "b") e pegará outro tipo de peça em DSI (transição "c"). O *token* é incrementado para 5, depois para 6 e 7, até que a peça seja depositada em MS3 (*token* 7). Em resumo, MS7 poderia processar a peça em paralelo, mas está em um evento com *token* individualizado. Neste caso, o cronograma não é bloqueante, mas desviará o controle da sequência ótima de produção da peça, causando atrasos.

O filtro de podagem evitará o desvio paralelizando e priorizando a execução das operações de eventos relacionados a recursos de *buffer* nulo e separando eventos contíguos de *buffer* limitado. A Figura 15-B explica as operações do filtro de podagem. Para o primeiro tipo de peça, o robô de VGR executará sequencialmente os eventos com *tokens* de 0 a 3, até depositar a peça

na estação MS7. O robô de VGR retornará descarregado para pegar o outro tipo de peça em DSI (transições "b" e "c") e a moverá para DSC (*token* 4) e NFC (*token* 5). Com o incremento do *token* para 6, NFC será priorizado (por ser recurso de *buffer* nulo) e processará a peça que depois será depositada em MS3. Em seguida, o robô de VGR é posicionado na estação MS7 (transição "d"), onde aguardará o processamento da peça (*token* 6) para depois movê-la para a próxima estação (MS1). Os demais eventos seguirão o processo padrão de controle de produção realizado pelo MAS, de acordo com o cronograma reajustado.

O filtro de podagem é apresentado no Algoritmo 6. E é a lista de eventos que representa o cronograma, e D é a lista de agentes associados a recursos de *buffer* limitado ou transitório limitado. Primeiramente, a função $check\_method()$ (linha 39) percorrerá os eventos do cronograma para identificar se o agente do evento pertence a D e, em caso afirmativo, se o evento anterior não pertence a D (linha 42). Quando esta condição é satisfeita, a função determina se o produto ou peça ($E_{i_{[job]}}$) identificado no evento atual é o mesmo do evento anterior ($E_{i-1_{[job]}}$) (linha 43). O evento atual é armazenado na lista L (linha 44), assim como outros eventos do cronograma classificados nas mesmas condições. Por fim, se a lista L contiver eventos, o cronograma de produção deve ser reajustado (linhas 48 e 49).

A função $exec\_pruning()$ reajusta o cronograma de produção (linha 26). Para cada evento carregado em L, a função obtém o respectivo índice (idx) a partir da posição do evento no cronograma original (linha 28). Em seguida, a função percorre o restante do cronograma verificando cada índice obtido para identificar o próximo evento de mesmo job ($E_{i_{[job]}}$) que o evento em análise (L_d) (linhas 29 e 30). Se encontrado, o evento L_d é inserido no cronograma, na posição definida por k (linha 31), e o evento antigo é deletado (linha 32). Por fim, o método chama a função $update\_schedule()$ para atualizar os *starts* ($E_{i_{[start]}}$) e *tokens* ($E_{i_{[token]}}$) em cada evento do cronograma modificado (linha 37).

4.2.3 Filtro de sobreposição

Os filtros anteriores podem evitar impasses e otimizar alguns cronogramas preditivos, mas não podem evitar sobreposições nos recursos de fabricação. Neste estudo, a sobreposição é entendida como uma tentativa de processar um produto em um recurso já ocupado por outro. O terceiro filtro, chamado de "filtro de sobreposição", é capaz de identificar sobreposições e evitá-las trocando a posição dos eventos causadores de impasses no cronograma de produção.

A Figura 16 explica o funcionamento do filtro de sobreposição para um cronograma

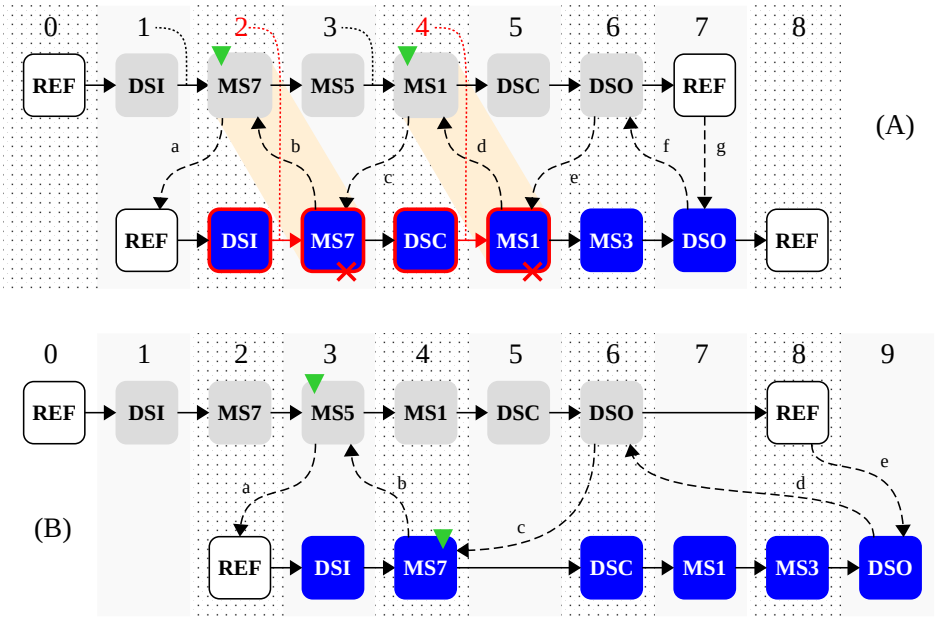
Algorithm 6 *Pruning filter*

```
1: E ← [E_0, ..., E_i, ..., E_f]                          ▷ cronograma de produção E
2: D ← [4, 7, 8, 9, 10, 11, 12, 13, 14, 15]               ▷ recursos de buffer limitado
3: function UPDATE_SCHEDULE()                              ▷ atualiza valores de start e token no cronograma
4:     E_0[start], E_0[token] ← 0
5:     ctl ← [ ]
6:     for (i ← 0,|E|) do
7:         if (E_i[job] = E_{i+1}[job]) then               ▷ eventos sequenciais de mesmo job
8:             E_{i+1}[start] ← E_i[start] + E_i[duration]
9:             E_{i+1}[token] ← E_i[token] + 1
10:            ctl ← [ ]
11:        else if (E_i[job] ! = E_{i+1}[job]) then        ▷ eventos sequenciais de jobs diferentes
12:            if (E_{i+1}[job] ∈ ctl) then
13:                E_{i+1}[start] ← E_i[start] + E_i[duration]
14:                E_{i+1}[token] ← E_i[token] + 1
15:                ctl ← [ ]
16:            else
17:                E_{i+1}[start] ← E_i[start]
18:                E_{i+1}[token] ← E_i[token]
19:                ctl ← append(E_i[job])
20:                ctl ← append(E_{i+1}[job])
21:            end if
22:        end if
23:    end for
24:    return E
25: end function
26: function EXEC_PRUNING(L)                               ▷ função que executa a podagem
27:    for (d ← 0,|L|) do
28:        idx ← E.index(L_d)                              ▷ idx é índice de um evento salvo em L
29:        for (k ← idx + 1,|E|) do                        ▷ percorre o cronograma a partir de idx + 1
30:            if (L_d[job] = E_k[job]) then               ▷ se job do evento de L igual a job de um evento em E
31:                E.insert(k, L_d)                        ▷ insere o evento de L na posição k
32:                E.remove(L_d)                           ▷ remove a primeira incidência do evento em E
33:                break
34:            end if
35:        end for
36:    end for
37:    return update_schedule(L)                           ▷ chama a função de atualização do cronograma
38: end function
39: function CHECK_METHOD()                                ▷ função que verifica contiguidade
40:    L ← [ ]
41:    for (i ← 0,|E|) do
42:        if (E_i[agent] ∈ D) and (E_{i−1}[agent] ∋ D) then   ▷ verifica se está em D e se há contiguidade
43:            if (E_i[job] = E_{i−1}[job]) and (i > 0) then
44:                L ← append(E_i)                         ▷ salva eventos contíguos em L
45:            end if
46:        end if
47:    end for
48:    if (|L| > 0) then                                   ▷ se é maior que zero, foram encontrados bloqueios
49:        return exec_pruning(L)                          ▷ chama a função de podagem
50:    else
51:        return E
52:    end if
53: end function
```

com dois tipos de peças. O primeiro tipo de peça é fabricado de acordo com a sequência $REF \prec DSI \prec MS7 \prec MS5 \prec MS1 \prec DSC \prec DSO \prec REF$, e o segundo tipo com $REF \prec DSI \prec MS7 \prec DSC \prec MS1 \prec MS3 \prec DSO \prec REF$. Primeiro, o cronograma original deverá ser submetido aos filtros de serialização e de podagem, resultando no cronograma

mostrado na Figura 16-A.

Figura 16 – Funcionamento do filtro de sobreposição.



Fonte: Autoria própria.

No entanto, o cronograma reajustado contém impasses devido às sobreposições destacadas em $DSI \rightarrow MS7$ (*token* 2) e em $DSC \rightarrow MS1$ (*token* 4). Em $MS7$, a sobreposição ocorrerá porque o robô de VGR depositará o primeiro tipo de peça neste recurso (*token* 1) e depois buscará o segundo tipo de peça em DSI (transição "a", *token* 1), para também tentar depositá-la em MS7 (*token* 2). No entanto, $MS7$ estará ocupada com o primeiro tipo de peça e o sistema entrará em *deadlock* devido à espera circular. Da mesma forma, a sobreposição também ocorrerá em $MS1$, que estará ocupada com o primeiro tipo de peça (transição "c", *token* 3) quando o robô tentar depositar a outra peça neste mesmo recurso (*token* 4). O filtro de sobreposição verifica a possibilidade de sobreposição nos recursos do FMS e gera um mapeamento de transição de estados para o cronograma analisado.

A Tabela 12 lista os índices dos eventos que se sobrepõe e o mapeamento realizado para o cronograma mostrado na Figura 16-A. O mapeamento gerado mostra as transições de sobreposição (identificadas por "S"), transições para eventos com recursos que estarão ocupados com peças (especificadas por "*"), os IDs de peças (entre colchetes, [0] e [1]) e os índices de sobreposição no cronograma (identificados em idx). A estratégia do filtro de sobreposição é localizar o índice do evento que possui o mesmo ID de agente e precede o evento de sobreposição (índice chamado wex), para trocar a sua posição e evitar o impasse. O evento de índice wex é inserido imediatamente após o evento de mesmo job que antecede wex no cronograma (um

índice chamado fdx). Essa estratégia de troca de posições preserva as sequências de operação originalmente definidas para a fabricação das peças.

Tabela 12 – Mapeamento das transições de estado e índices de sobreposição.

```
»»» generate overlay mapping:
REF -> DSI [0]  [0, 5, 0, 0, 0] -> [5, 5, 1, 0, 1]
DSI -> MS7 [0]* [5, 5, 1, 0, 1] -> [10, 5, 13, 0, 2]
MS7 -> REF [a]
REF -> DSI [1]  [5, 5, 0, 1, 1] -> [10, 5, 1, 1, 2]
DSI -> MS7 [1]S [10, 5, 1, 1, 2] -> [15, 5, 13, 1, 3]
MS7 -> MS5 [0]  [10, 5, 13, 0, 2] -> [15, 5, 11, 0, 3]
MS5 -> MS1 [0]* [15, 5, 11, 0, 3] -> [20, 5, 7, 0, 4]
MS1 -> MS7 [b]
MS7 -> DSC [1]  [15, 5, 13, 1, 3] -> [20, 5, 2, 1, 4]
DSC -> MS1 [1]S [20, 5, 2, 1, 4] -> [25, 5, 7, 1, 5]
MS1 -> DSC [0]  [20, 5, 7, 0, 4] -> [25, 5, 2, 0, 5]
DSC -> DSO [0]  [25, 5, 2, 0, 5] -> [30, 5, 14, 0, 6]
DSO -> MS1 [c]
MS1 -> MS3 [1]  [25, 5, 7, 1, 5] -> [30, 5, 9, 1, 6]
MS3 -> DSO [1]  [30, 5, 9, 1, 6] -> [35, 5, 14, 1, 7]
DSO -> DSO [d]
DSO -> REF [0]  [30, 5, 14, 0, 6] -> [35, 5, 0, 0, 7]
DSO -> REF [1]  [35, 5, 14, 1, 7] -> [40, 5, 0, 1, 8]
   overlap idx: [6, 10]
```

Fonte: Autoria própria.

A Figura 16-B mostra o cronograma reajustado e livre de sobreposições. O filtro de sobreposição reajustará o cronograma para que a transição “a”, no primeiro tipo de peça, não ocorra no recurso que também é compartilhado com o segundo tipo de peça (recurso $MS7$). Consequentemente, quando o robô de VGR movimentar o segundo tipo de peça (sequência em azul), $MS7$ estará livre. Adicionalmente, a sobreposição que ocorreria em $MS1$ também será corrigida com a troca de posição do evento para o segundo tipo de peça (*token* 7).

O filtro de sobreposição é apresentado no Algoritmo 7. R é uma lista com recursos de fabricação usados no mapeamento, B é uma lista de agentes associados a recursos de *buffer* limitado e A é um alfabeto para as transições (linha 1). M, L e K são variáveis para, respectivamente, salvar dados de mapeamento, dados de índices de sobreposição e nomes de recursos usados no controle de decisão (linha 1). Inicialmente, o filtro de sobreposição usa a função $check\_method()$ (linha 27) para verificar sobreposições. Assim, os índices de eventos que se sobrepõe são armazenados em L (linha 51). As linhas 30 a 47 referem-se ao mapeamento das transições para os recursos de *buffer* limitado ocupados com peças (linha 38), transições de processamento normal (linha 41) e transições de sobreposição (linha 50). Em seguida, é disponibilizado o mapeamento (linhas 57 e 58) e os índices de sobreposição armazenados em L (linha 60). Se L não for nulo, significa que sobreposições foram identificadas no mapeamento.

Algorithm 7 *Overlay filter*

$R \leftarrow [REF, ..., DSO]$; $B \leftarrow [7, 8, ..., 12, 13]$; $A \leftarrow [ASCII\_letters]$; $a \leftarrow 0$; $M, L, K \leftarrow [\,]$
function FIX_OVERLAY() ▷ função que corrige as sobreposições
  for $(i \leftarrow 0, |L|)$ **do**
    for $(j \leftarrow L_i - 1, 0)$ **do**
      if $(E_{L_{i_{[agent]}}} = E_{j_{[agent]}})$ **then**
        $wex \leftarrow E.index(E_j)$ ▷ obtém índice anterior ao evento de sobreposição
        break
      end if
    end for
    for $(j \leftarrow L_i + 1, |E|)$ **do**
      if $(E_{L_{i_{[agent]}}} = E_{j_{[agent]}})$ **then**
        $wex \leftarrow E.index(E_j)$ ▷ obtém índice posterior ao evento de sobreposição
        break
      end if
    end for
    for $(e \leftarrow wex - 1, 0)$ **do**
      if $(E_{wex_{[agent]}} = E_{e_{[agent]}})$ **then**
        $fdx \leftarrow E.index(E_e)$ ▷ obtém índice do evento que antecede wex
        break
      end if
    end for
    $aux \leftarrow E_{wex}$
    $E.remove(E_{wex})$ ▷ remove o evento do índice wex
    $E.insert(fdx + 1, aux)$ ▷ insere o evento de aux após o evento de índice fdx
  end for
end function
function CHECK_METHOD() ▷ função que verifica sobreposições
  for $(e \leftarrow 0, |E|)$ **do**
    for $(i \leftarrow e, |E|)$ **do**
      if $(E_{e_{[job]}} = E_{i+1_{[job]}})$ **then** ▷ verifica se o evento atual e o próximo são do mesmo job
        $src \leftarrow R[E_{e_{[agent]}}]$ ▷ src é o evento de origem
        $dst \leftarrow R[E_{i+1_{[agent]}}]$ ▷ dst é o evento de destino
        if $(src \in K)$ **then**
          $K.remove(src)$
        end if
        if $(dst \ni K)$ **then**
          if $(E_{i+1_{[agent]}} \in B)$ and $(i - e > 0)$ **then** ▷ verifica se dst é *buffer* limitado
            $M \leftarrow append(src + \text{“->”} + dst + \text{“*”})$
            $K \leftarrow append(dst)$ ▷ salva dst na lista de recursos ocupados K
          else
            $M \leftarrow append(src + \text{“->”} + dst + \text{“ ”})$
          end if
          if $(i - e > 0)$ **then**
            $src \leftarrow R[E_{i+1_{[agent]}}]$
            $dst \leftarrow R[E_{e+1_{[agent]}}]$
            $M \leftarrow append(src + \text{“->”} + dst + \text{A[a]})$ ▷ M é usado para mapeamento das transições
            $a \leftarrow a + 1$
          end if
        else ▷ se dst já está em K, significa sobreposição
          $M \leftarrow append(src + \text{“->”} + dst + \text{“S”})$
          $L \leftarrow append(E.index(E_{i+1}))$ ▷ salva índice do evento de sobreposição
        end if
        break
      end if
    end for
  end for
  for $(i \leftarrow 0, |M|)$ **do**
    $print(M[i])$
  end for
  $print(\text{“overlay idx: ”}, L)$
  if $(|L| > 0)$ **then** ▷ se tamanho de L é maior que zero
    return $fix\_overlay(L)$ ▷ chama a função de correção da sobreposição
  else
    return E
  end if
end function

A função $fix\_overlay()$ reajusta os cronogramas de produção que contêm sobreposições (linha 2). Primeiro, a função obterá o índice do evento de mesmo ID de agente que o evento de sobreposição e o armazenará em wex (linhas 4 a 15). Em seguida, a função obterá o índice do evento anterior a wex e o salvará em fdx (linhas 16 a 21). O evento de índice wex é deslocado e inserido imediatamente após o evento de índice fdx (linhas 22–24). Essa sequência de código é executada para cada índice de sobreposição contido em L. Por fim, o cronograma é reajustado e verificado sucessivamente, repetindo as funções $check\_method()$ e $fix\_overlay()$, até que esteja livre de sobreposições. A cada rodada de reajustes, os filtros de serialização e de podagem também são aplicados para evitar possíveis *deadlocks*. Ao final de todos os reajustamentos, o cronograma poderá ser enviado para o MAS.

4.3 AVALIAÇÃO GERAL

Um artigo relatando as modificações efetuadas na plataforma de experimentação, para operar como um FMS controlado por ordem, foi publicado em congresso. Outro artigo submetido em revista incluiu experimentos com os filtros de reajustamento e o MAS proposto. Os resultados mostraram que a solução pôde evitar com sucesso os problemas de impasses discutidos no estudo. Mesmo com a serialização de alguns eventos, o *makespan* obtido é semelhante ao comprimento ótimo calculado pelo algoritmo preditivo, pois o número de transições nos cronogramas reajustados é menor. Além disso, o número de eventos nos cronogramas permanece o mesmo, nenhum evento requer maior tempo de processamento no sistema (a duração dos eventos não é alterada), nenhum *buffer* extra é necessário e nenhuma ação depende de reescalonamento para garantir cronogramas de produção funcionais e livres de impasses.

Tabela 13 – Eficiência dos filtros de reajustamento.

	Sch.	**Completed schedules**			**Efficiency (%)**		
		FJS	**S+P**	**OVF**	**FJS**	**S+P**	**OVF**
Expt.1	8	0	6	8	✗ 0	✓ 75	✓ 100
Expt.2	3	0	3	3	✗ 0	✓ 100	✓ 100
Expt.3	9	0	0	9	✗ 0	✗ 0	✓ 100
Expt.4	10	0	10	10	✗ 0	✓ 100	✓ 100
				AGV.	**0.00**	**68.75**	**100.00**

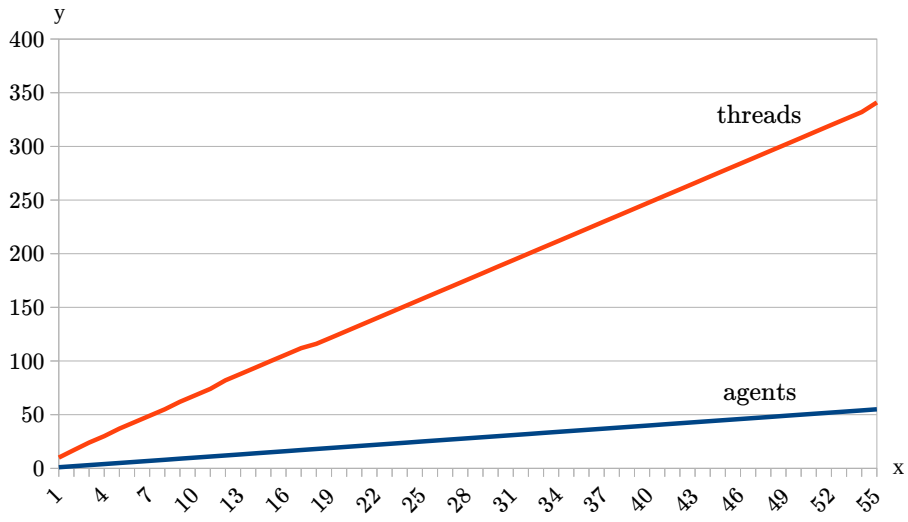
Fonte: Autoria própria.

A Tabela 13 resume dados do artigo (SOUSA; OLIVEIRA, 2023) sobre a eficiência dos métodos propostos para um conjunto de 30 cronogramas de produção distribuídos em quatro experimentos. A última linha da tabela apresenta a média dos percentuais de eficiência

dos algoritmos. No primeiro experimento (Expt.1), os filtros de serialização e podagem (S+P) garantiram que o MAS concluísse 75% dos cronogramas reajustados, ou seja, 6 cronogramas. Em comparação, com o FJS o percentual foi 0%, todos os oito cronogramas de produção do experimento não puderam ser concluídos pelo MAS devido a impasses. Nos demais experimentos, o percentual de 0% para os cronogramas gerados pelo FJS se repetiu devido à incapacidade do algoritmo de evitar impasses. No terceiro experimento (Expt.3), a eficiência dos filtros de serialização e de podagem foi de 0% porque os cronogramas reajustados ainda continham *deadlocks* por sobreposição (idem para dois cronogramas não completados do Expt.1). Por fim, a combinação dos filtros de serialização e de podagem, com o filtro de sobreposição (representado na tabela por OVF) garantiu a conclusão de 100% dos cronogramas avaliados nos quatro experimentos.

O controle de consenso da arquitetura MAS proposta garante a convergência em tempos pré-definidos nos cronogramas, o que possibilita que os agentes deduzam o estado atual do sistema de acordo com o percentual de eventos executados a cada troca de *token*. Este sincronismo no MAS proporciona previsibilidade de produção e estabilidade para o FMS. Além disso, a camada de comunicação (incluindo o consenso) é idêntica em todos os agentes, favorecendo a escalabilidade e a adaptabilidade da abordagem a outros ambientes. Para fins de informação, a escalabilidade de carga da arquitetura MAS foi atestada com agentes virtuais (Figura 17).

Figura 17 – Escalabilidade da arquitetura MAS.



Fonte: Autoria própria.

Em média, um único agente no sistema executa entre seis e sete *threads*. Com dois agentes, cada um executa de 13 a 14 *threads*; com três agentes, 20 *threads* cada; com quatro agentes, 26 *threads* cada; e assim por diante. Cinquenta e cinco agentes virtuais foram executados

em um computador Intel Core i5-8250U 1.60GHz com 8GB de RAM, resultando em 341 *threads* por agente, ou 18.755 *threads* no total. À medida que o número de agentes no sistema aumenta, o número de *threads* aumenta linearmente (com um coeficiente angular maior, tecnicamente) devido ao aumento de conexões entre agentes. Como os agentes são implementados com *threads* para cada tópico ROS da camada de comunicação e para o controle, o número total de agentes no MAS é limitado apenas pelos recursos do computador que hospeda os agentes virtuais. Porém, em um ambiente de produção real, a carga computacional será distribuída no hardware dos recursos de produção, tornando esta limitação irrelevante. Neste caso, a largura de banda provavelmente será um recurso limitante. Mas há uma variedade de soluções que permitem a integração de sistemas baseados em ROS em ambientes industriais, como o pacote de software ROS-Industrial[1] que contém bibliotecas, ferramentas e drivers para hardware industrial. Além disso, existem conversores para integração em redes industriais (por exemplo, redes PROFINET) e *gateways* que permitem acesso rápido e fácil a ecossistemas industriais de IoT.

4.3.1 Complexidade dos algoritmos

A complexidade de tempo representa o tempo necessário para um algoritmo executar em função do comprimento (quantidade de dados ou tamanho) de sua entrada. Informações sobre a complexidade de tempo dos algoritmos propostos são essenciais para avaliar o emprego da abordagem em outros ambientes de produção. A Tabela 14 apresenta dados sobre a complexidade temporal dos agentes e filtros de reajustamento desenvolvidos. As notações Ω e O foram utilizadas para expressar o limite inferior (melhor caso) e limite superior (pior caso) dos algoritmos.

Tabela 14 – Complexidade de tempo dos algoritmos propostos.

		Melhor caso	**Pior caso**
Agentes	***Signaling***	$\Omega(n+7)$	$O(n^2+3n+13)$
	Synchrony	$\Omega(n+16)$	$O(n^2+5n+21)$
	Notification	$\Omega(5)$	$O(n+7)$
	Broadcast	$\Omega(2)$	$O(2)$
Filtros	**Serialização**	$\Omega(2n^2+10n+28)$	$O(2n^2+14n+46)$
	Podagem	$\Omega(n^2+4n+27)$	$O(n^2+8n+33)$
	Sobreposição	$\Omega(2n^2+9n+52)$	$O(2n^2+12n+61)$

Fonte: Autoria própria.

A complexidade de tempo dos agentes refere-se a um único agente, pois o modelo de agente é padrão. Porém, o cálculo da complexidade se concentra na camada de comunicação da

[1] ROS-Industrial: https://rosindustrial.org/

arquitetura MAS. As camadas de controle e física, que tem relação com os recursos que estão associados com cada agente, foram omitidas do cálculo. Declarações condicionais, atribuições e operações simples são calculadas em tempos constantes. Outras operações mais complexas foram calculadas de acordo com a complexidade de tempo definida nos documentos do CPython atual. Cada instrução de troca de mensagens relacionada com o consenso também foi computada com tempo constante $O(1)$. Na Tabela 14, a soma das complexidades calculadas para "*synchrony*", "*signaling*" e "*notification*" representa a complexidade de tempo do consenso. Por fim, são apresentadas as complexidades de tempo dos filtros de serialização, podagem e sobreposição. No filtro de sobreposição, a complexidade de tempo da recursão não é computada, pois depende do número de vezes que o método chama a si mesmo. É fundamental ressaltar que, como os algoritmos são executados antes do envio dos cronogramas ao MAS, o tempo gasto no reajuste não influencia no desempenho do FMS.

4.4 CONCLUSÕES DO CAPÍTULO

Este capítulo apresentou uma arquitetura multiagente para sistemas flexíveis de manufatura, enfatizando a aplicabilidade do consenso em problemas de controle de manufatura. Na literatura, os algoritmos de consenso raramente foram adaptados para lidar com o controle da produção ou ainda não foram implementados em sistemas flexíveis de manufatura com a profundidade e o detalhamento necessários para permitir a portabilidade para outros ambientes de produção, aprimoramentos e a incorporação em soluções comerciais. Neste estudo, três filtros para o reajustamento de cronogramas preditivos gerados por um algoritmo de FJS foram desenvolvidos e avaliados. Experimentos comprovaram que algoritmos tradicionais de escalonamento preditivo, combinados com MAS e filtros para reajustamento de cronogramas possibilitam uma produção flexível e livre dos impasses discutidos no capítulo.

A reprodução da abordagem em outros cenários é simples, mas requer algumas adaptações. Para a utilização dos filtros de reajustamento é necessário indicar quais são os recursos de *buffer* nulo (variável B, nos Algoritmos 5 e 7) e os de *buffer* limitado (variável D, no Algoritmo 6), bem como definir uma nomenclatura para os recursos disponíveis no FMS (variável R, no Algoritmo 7). Os cronogramas de produção também deverão ser disponibilizados para os filtros conforme o modelo apresentado no Exemplo 2 (ver Capítulo 3). Para a arquitetura MAS, a camada de comunicação e o consenso podem ser replicados sem adaptações, para agentes utilizando o *framework* ROS. O trabalho de adaptação mais significativo se concentrará nas

camadas de controle e física, pois serão específicas para os recursos de fabricação do novo cenário. Algumas soluções disponíveis, como o OpenPLC[2], Raspberry Pi e derivados (p. exe., UniPi e PiExtend), ou ainda outros controladores com Linux embarcado que suportem a Python e rospy[3], podem facilitar essa integração.

Na literatura não há informações sobre FMSs operando com MAS e métodos semelhantes aos propostos neste estudo para fins de comparação. Algumas propostas limitam-se a resultados teóricos, com simplificações que inviabilizam a portabilidade das soluções para ambientes reais ou mesmo reproduzi-las em outros estudos. A capacidade de *buffer* dos recursos, bem como limitações relacionadas à movimentação de produtos no sistema produtivo, são determinantes para a aplicabilidade das soluções. Essas questões não podem ser simplificadas nas pesquisas, pois não refletem a realidade dos sistemas produtivos em operação. Entretanto, alguns problemas são difíceis de resolver no reajustamento dos cronogramas, como situações imprevistas durante a produção, como ordens de urgência e falhas em equipamentos. Nestes casos, o reescalonamento da produção pode ser o foco da pesquisa, que pode ser complementar à abordagem proposta neste estudo, e vice-versa.

[2] OpenPLC: https://openplcproject.com/
[3] Biblioteca cliente Python para ROS: http://wiki.ros.org/rospy

5 ESCALONAMENTO COM RESTRIÇÕES E URGÊNCIAS

A previsibilidade na produção é crucial para as indústrias poderem atender as suas demandas de maneira ainda mais eficiente, com custos menores e em níveis adequados. Entretanto, o escalonamento preditivo convencional em ambientes de produção sujeitos à restrições e urgências compromete a previsibilidade, especialmente quando recursos com limitações de *buffer* e de transporte do ambiente são compartilhados. Este estudo apresenta um algoritmo de escalonamento preditivo para FMSs com restrições de *buffer* e de transporte. Os cronogramas gerados são livres de *deadlocks* e dois tipos de consenso multiagente são implementados para suportar a inserção de cronogramas de ordens urgentes no sistema. O consenso por tempo finito pré-definido lida com cronogramas de produção de ordem normal, enquanto que o consenso de grupo ou cluster garante a execução de cronogramas relacionados com as ordens de urgência. Um método baseado em janelas de tempo é utilizado para validar os momentos de inserção das ordens de urgência no sistema, de forma que não ocorram *deadlocks* por sobreposição de tarefas em recursos ocupados com os cronogramas de ordem normal.

5.1 CARACTERÍSTICAS DO TRABALHO

Apesar de problemas de *job-shop scheduling* terem sido amplamente estudados, ainda faltam soluções que ofereçam um tratamento mais abrangente relacionado à capacidade de *buffer* e de transporte (ou roteamento) dos ambientes de produção. Por vários motivos discutidos nos capítulos anteriores, entende-se que é adequado que restrições de *buffer* e de transporte, aqui definidas como BTCs (*buffer and transportation constraints*), sejam tratadas durante o escalonamento. Existem processos de fabricação que dependem do sistema de transporte/roteamento, e que o ocupam por certo tempo para efetuar determinadas operações, seja este sistema formado por braços robóticos, pontes móveis, AGVs (*automated guided vehicles*), etc. FMSs que utilizam cronogramas preditivos gerados por processos de escalonamento convencional são sensíveis a BTCs.

Em geral, o objetivo do escalonamento para FMSs é utilizar o máximo dos recursos disponíveis para obter um custo de fabricação competitivo, tendo em vista o alto custo de implantação do sistema (MAREDDY *et al.*, 2022). Entretanto, algoritmos de escalonamento tradicionais que maximizam a utilização dos recursos e calculam *makespan* mínimo (*Cmax*) como medida

de desempenho teórica apropriada, acabam retornando valores inadequados de *makespan* no escalonamento preditivo sujeito a BTCs. Isso ocorre porque as transições do sistema de transporte em cronogramas com múltiplos produtos são frequentes, ou seja, o sistema normalmente move-se sem carga (vazio) sempre que precisa lidar com o roteamento de um produto diferente. Em outras palavras, o escalonamento tradicional considera que o roteamento é prontamente aplicado a cada operação de produto concluída, o que na prática não é verdade quando o sistema de transporte é compartilhado para a fabricação de múltiplos produtos. Entretanto, a obtenção de valores inadequados de *makespan* comprometem a previsibilidade do FMS para atendimento às demandas de produção.

Este estudo apresenta um algoritmo para problemas de escalonamento de *job-shop* flexível com restrições de *buffer* e de transporte, denominado FJS-BTC (*flexible job-shop scheduling with buffer and transportation constraints*). Problemas de escalonamento em FMSs com produção controlada por ordem, onde os produtos finais dependem de operações em uma sequência específica de máquinas (*multi-machine multi-product system*), podem ser caracterizados como um problema de FJS-BTC.

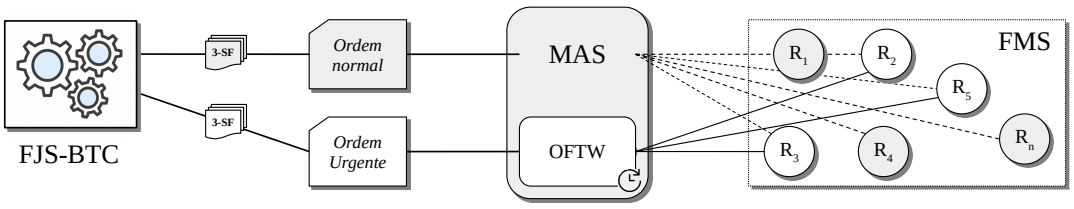
5.1.1 Problemas de FJS-BTC

Um problema de *job-shop* flexível com restrições de *buffer* e de transporte (FJS-BTC) envolve um conjunto de máquinas $M = \{M_1, ..., M_m\}$ (recursos), e um conjunto de *jobs* $J = \{J_1, ..., J_n\}$ (tarefas). Cada job J_i é composto por k operações $O_{i,1}, ..., O_{i,k}$ e cada operação $O_{i,h}$ requer uma máquina M_h para a sua execução. Em um cronograma factível, as operações de cada job devem ser executadas em uma sequência predefinida; operações de um mesmo job não podem executar em paralelo. Cada máquina pode executar no máximo uma operação por vêz e a recirculação de $jobs$ na mesma máquina é permitida. Em um problema de FJS-BTC há Z máquinas de *buffer* nulo ininterrupto e L máquinas de *buffer* limitado transitório, tal que Z e L são subconjuntos de M. *Buffer* nulo ininterrupto se refere às máquinas ou recursos de fabricação com capacidade zero de *buffer*, onde as operações devem ser ininterruptas nos recursos ou entre eles devido a ocupação do sistema de transporte (p. ex., posicionamento de peças com braço robótico ou ponte móvel sobre o recurso). *Buffer* limitado transitório significa que os próprios recursos servem de *buffer* (com capacidade um) e retêm os $jobs$ até que os recursos subsequentes sejam liberados. Um job é finalizado depois que todas as suas operações forem concluídas. O tempo de conclusão de operações individuais e paralelas ocorre em intervalos pré-definidos,

considerando a operação de maior tempo em cada grupo de operações. Neste estudo o tempo de conclusão de J_i é definido pelo somatório dos intervalos relacionados às operações individuais e paralelas do job, enquanto que o tempo total de conclusão de todas as tarefas (*makespan*) equivale ao somatório dos tempos de conclusão de cada grupo de operações.

A continuação deste estudo se concentra no suporte à ordens de urgência, uma característica desejável para FMSs com produção controlada por ordem. Entretanto, a inserção de ordens de urgência está condicionada à disponibilidade dos recursos de produção e precisa ser avaliada em tempo de execução. Logo, este estudo também propõe o aproveitamento do consenso multiagente para garantir que as ordens de urgência, que são prioritárias às ordens normais, sejam inseridas no FMS sem que ocorram sobreposições ou outros impasses relacionados a BTCs. Um algoritmo chamado OFTW é utilizado para validar os momentos de inserção das ordens de urgência no sistema. A utilização do consenso multiagente no apoio às decisões de inserção de ordens de urgência ainda não foi explorada em FMSs sujeitos a BTCs. A Figura 18 apresenta uma visão geral deste estudo.

Figura 18 – Visão geral da proposta envolvendo FJS-BTC, MAS, OFTW e FMS.



Fonte: Autoria própria.

O algoritmo FJS-BTC gera cronogramas de produção livres de *deadlocks* vinculados à ordens normais ou urgentes. Os cronogramas das ordens normais são enviados ao MAS e executados sequencialmente por agentes associados a algum recurso de produção do FMS. Quando o MAS recebe uma ordem urgente, o algoritmo OFTW avalia o melhor momento de inserir o cronograma para evitar *deadlocks* no FMS. Quando a ordem urgente é concluída, a programação regular é retomada e a produção continua sem interrupções. O MAS usa dois tipos de consenso para garantir que os agentes cooperem nas tarefas de produção, um para pedidos regulares e outro para ordens urgentes.

5.2 ABORDAGEM PROPOSTA

Esta seção apresenta o algoritmo de escalonamento proposto, as atualizações efetuadas no modelo de agente para suportar controle de consenso por tempo finito pré-definido e por grupo/*cluster*, e o algoritmo para validação da inserção de ordens de urgência em FMSs.

5.2.1 Algoritmo FJS-BTC

O FJS-BTC é um algoritmo de escalonamento preditivo, baseado em informações completas, que a partir de um modelo de produção obtém um cronograma livre de impasses relacionadas a recursos de *buffer* nulo ininterrupto e de *buffer* transitório limitado (BTCs). Um modelo de produção é formado por sequências de tuplas (M_h, t), de forma que cada sequência representa uma tarefa (job) e cada tupla da sequência identifica a máquina M_h e o tempo t de processamento de máquina (ou recurso) disponibilizado para o job. O algoritmo FJS-BTC utiliza cinco métodos de obtenção de composições que determinam a ordem das sequências de tuplas ($jobs$) no modelo de produção:

- Permutação simples (*easy permutation*): consiste em formar composições ordenadas de n elementos, de um conjunto finito de elementos distintos que não se repetem. A quantidade de composições distintas que podem ser formadas é dada por $P_n = n!$, tal que um conjunto de n elementos é igual a $n!$ (lê-se n fatorial). A permutação simples é um tipo de regra de contagem estudada em Análise Combinatória.

- Permutação circular (*circular permutation*): consiste em formar composições de n elementos distintos em ordem circular (formando uma circunferência). Não há um elemento de início e um fim nas composições, a posição é definida a partir do momento em que um elemento assume qualquer uma das posições do círculo. O número de maneiras de organizar n elementos no círculo é dado por $P_n = (n-1)!$.

- Deslocamento simples (*easy shift*): consiste em formar composições de n elementos deslocando um dos elementos para a direita ou para a esquerda por um número de vezes igual à quantidade de elementos do conjunto, ou seja, $P_n = n$. No algoritmo de FJS-BTC a composição inicial para a matriz de eventos é definida aleatoriamente, e o primeiro elemento da composição é deslocado para formar as novas composições.

- Deslocamento cíclico (*cyclic shift*): consiste em formar composições de n elementos deslocando cada elemento por um número de vezes igual a n, sem repetir as composições. Dado uma composição inicial de tamanho n, o número de composições distintas que podem ser obtidas é definido por $P_n = (n \times n) - n$. No algoritmo FJS-BTC, a composição inicial é disponibilizada em ordem crescente.

- Matriz aleatória (*random array*): consiste em formar composições de elementos aleatoriamente. A quantidade de composições obtidas é definida pelo produto de $n \times m$, onde n é o número sequências e m é o número de máquinas da maior sequência disponível no modelo de produção. Estatisticamente, o número de composições sem repetição equivale a 50% (aprox.) do total de composições obtidas pelo método. Portanto, a quantidade de composições é definida por $P_{n,m} = (n \times m)/2$.

Supondo que um modelo de produção tenha cinco sequências e a maior sequência do modelo seja composta por dez máquinas, cada método deverá obter:

- $P_5 = 5!$, ou 120 composições com a permutação simples.
- $P_5 = (5 - 1)!$, ou 24 composições com a permutação circular.
- $P_5 = n$, ou 5 composições com o deslocamento simples.
- $P_n = (n \times n) - n$, 20 composições com o deslocamento cíclico.
- $P_{5,10} = (5 \times 10)/2$, ou 25 composições (aprox.) na matriz aleatória.

No modelo de produção a ordem das sequências é organizada segundo cada composição e o modelo é carregado em uma matriz de eventos, de forma que cada composição define uma matriz distinta. Por exemplo, a Figura 19 apresenta a matriz de um modelo de produção inicial (composição $[J_0, J_1, J_2]$) e outras duas matrizes derivadas do modelo inicial, de acordo com as composições $[J_2, J_0, J_1]$ e $[J_1, J_0, J_2]$. A ordem das tuplas nas matrizes permanece inalterada, pois a diferença da matriz inicial para as demais está na posição das linhas.

A próxima ação do algoritmo de escalonamento é efetuar a transformação das matrizes. Para cada matriz, o FJS-BTC executa quatro operações: (1) selecionar a coluna de trabalho na matriz; (2) identificar as máquinas da coluna de trabalho; (3) detectar repetições de recursos na coluna; e (4) deslocar a sequência que provoca repetição (se for o caso). Estas quatro operações se repetem até que todas as colunas de trabalho da matriz sejam verificadas.

Figura 19 – Composições obtidas a partir da matriz de eventos inicial.

	M_h, t			
J_0	(0, 5)	(1, 5)	(8, 7)	(4, 5)
J_1	(0, 5)	(1, 5)	(7, 9)	(0, 5)
J_2	(0, 5)	(4, 5)	(1, 5)	(3, 5)

↗

J_2	(0, 5)	(4, 5)	(1, 5)	(3, 5)
J_0	(0, 5)	(1, 5)	(8, 7)	(4, 5)
J_1	(0, 5)	(1, 5)	(7, 9)	(0, 5)

matriz da ⟵ composição 1

↘

J_1	(0, 5)	(1, 5)	(7, 9)	(0, 5)
J_0	(0, 5)	(1, 5)	(8, 7)	(4, 5)
J_2	(0, 5)	(4, 5)	(1, 5)	(3, 5)

matriz da ⟵ composição 2

Fonte: Autoria própria.

A Figura 20 apresenta as matrizes das composições do exemplo anterior após o processo de transformação do FJS-BTC. Além de deslocar as sequências (célula a célula) a partir de cada coluna de trabalho, o algoritmo de escalonamento também insere células vazias (None) para igualar o número de células de cada sequência durante o redimensionamento da matriz. O tamanho da matriz aumenta devido à necessidade de deslocar as sequências para evitar a repetição de máquinas (ou recursos), pois a coluna de trabalho se refere às máquinas que executarão determinadas operações de $jobs$ paralelamente.

Figura 20 – Apresentação das matrizes de eventos após a transformação.

J_0	(0, 5)	(1, 5)	(8, 7)	(4, 5)	None	None	None
J_1	None	(0, 5)	(1, 5)	(7, 9)	(0, 5)	None	None
J_2	None	None	(0, 5)	None	(4, 5)	(1, 5)	(3, 5)

matriz ⟵ inicial

J_2	(0, 5)	(4, 5)	(1, 5)	(3, 5)	None	None	None
J_0	None	(0, 5)	None	(1, 5)	(8, 7)	(4, 5)	None
J_1	None	None	(0, 5)	None	(1, 5)	(7, 9)	(0, 5)

matriz da ⟵ composição 1

J_1	(0, 5)	(1, 5)	(7, 9)	(0, 5)	None	None
J_0	None	(0, 5)	(1, 5)	(8, 7)	(4, 5)	None
J_2	None	None	(0, 5)	(4, 5)	(1, 5)	(3, 5)

matriz da ⟵ composição 2

Fonte: Autoria própria.

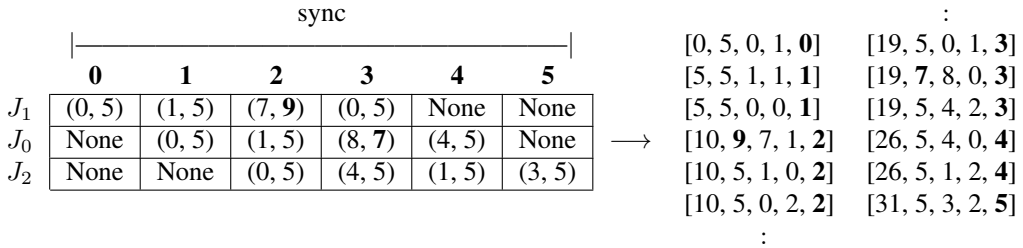
Como o algoritmo FJS-BTC obtém um determinado número de matrizes distintas, conforme o método de obtenção de composições utilizado, é preciso selecionar aquela matriz que oferece menor custo para o escalonamento. Para isso, o algoritmo obtém o maior tempo de processamento de cada coluna da matriz para somatório e obtenção do *makespan* teórico. A matriz com menor número de colunas e menor *makespan* teórico é eleita para a formação do cronograma de produção.

A próxima ação do algoritmo FJS-BTC consiste em gerar um cronograma de teste, a partir da matriz obtida, para a identificação de potenciais *deadlocks* devido às restrições de BTC mencionadas anteriormente. Um cronograma é formado por uma sequência de eventos E e cada evento é definido por $E_i = [start, duration, machine, job, sync]$. Esta definição de evento é equivalente àquela apresentada no Capítulo 3 (ver subseção 3.1.4).

O algoritmo FJS-BTC cria um evento para cada célula da matriz ótima, de forma que as tuplas (M_h, t) em cada coluna da matriz são configuradas com o mesmo tempo de início ($start$) e o mesmo índice de controle de sincronismo ($sync$). Entretanto, a cada avanço do algoritmo sobre as colunas da matriz, o tempo de início é incrementado com o valor da tupla de maior duração (t) da coluna anterior. Isso ocorre para conformidade ao BTC, pois o algoritmo deve considerar a relação entre a capacidade do sistema de transporte e os recursos de *buffer* nulo ininterrupto. Na prática, o sistema estará sincronizado quando a operação de maior tempo de processamento para o índice corrente ($sync$) finalizar, liberando o sistema de transporte. Portanto, durante a geração do cronograma de produção, o algoritmo FJS-BTC incrementa o índice de controle de sincronismo ($sync$) a cada avanço sobre as colunas da matriz. Por fim, o *makespan* teórico é resultado da soma do valor de $start$ da última coluna da matriz, com o maior valor de $duration$ da mesma coluna.

A Figura 21 exemplifica o processo de geração de um cronograma com o algoritmo FJS-BTC. Cada coluna da matriz está associada a um índice de controle de sincronismo, que vai de zero até cinco. O tempo de duração da maioria das operações é 5 segundos (neste exemplo), exceto para a tupla $(8, \mathbf{7})$ que é 7 segundos, e para a tupla $(7, \mathbf{9})$ que é 9 segundos. O cronograma possui 12 eventos, de E_0 a E_{11}. O atributo job indica o respectivo J_i da composição $[J_1, J_0, J_2]$. O *makespan* teórico para o cronograma gerado é de 36 segundos, calculado com a soma dos valores de $start$ e $duration$ $(31 + 5)$ do último evento do cronograma $[31, 5, 3, 2, 5]$; se houvessem eventos paralelos (mais de uma tupla na última coluna da matriz), se consideraria o maior valor de $duration$ para o cálculo do *makespan*.

Figura 21 – Exemplo de formação do cronograma de produção.

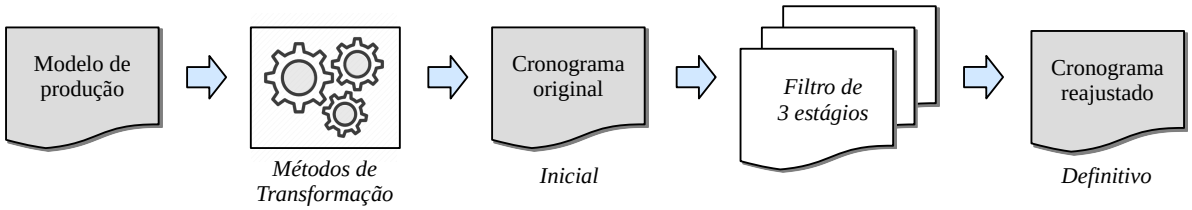


Fonte: Autoria própria.

Em resumo, um modelo de produção é utilizado pelo algoritmo FJS-BTC para gerar um cronograma inicial que, em seguida, é submetido a um filtro de três estágios de reajustamento para tratar impasses por BTC e formar o cronograma definitivo (Figura 22). O filtro de três estágios é uma adaptação dos filtros apresentados no estudo anterior (Capítulo 4, seção 4.2).

Por fim, concluída a sequência de processos do FJS-BTC, o cronograma reajustado e livre de impasses poderá ser enviado ao MAS para a execução da produção.

Figura 22 – Sequência de processos do algoritmo FJS-BTC.



Fonte: Autoria própria.

A próxima seção apresenta as atualizações efetuadas no sistema multiagente para suportar, em tempo de execução, as ordens de produção urgentes geradas pelo algoritmo FJS-BTC.

5.2.2 Atualizações no MAS

Para que os agentes suportem a inserção de ordens de urgência em um FMS em produção, o modelo de agente da arquitetura MAS proposta no estudo anterior (Capítulo 4, subseção 4.1.1) precisou ser atualizado. As atualizações envolvem o suporte à notificações de urgência, checagem de prioridade, troca de contextos e a adaptação dos agentes ao tipo de consenso; por tempo finito pré-definido e grupo. Da mesma forma que o estudo anterior, o modelo de agente atualizado mantém quatro tópicos em comum para o controle de consenso: "*synchrony*", "*signaling*", "*broadcast*" e "*notification*". Estes tópicos permaneces associados aos seus respectivos algoritmos, que são executados pelos agentes em *threads*. Como as atualizações implementadas foram significativas, os algoritmos atuais são apresentados e explicados

O código relacionado ao tópico *synchrony* é apresentado no Algoritmo 8. A função $publish\_sync$ atualiza o índice de controle de sincronismo ($sync[s]$) do agente para um cronograma de ordem normal ($s = 0$) ou para um cronograma de ordem urgente ($s = 1$). Cada agente deve estar associado a um ou mais recursos de produção do ambiente de manufatura. Os recursos são identificados por IDs que devem estar listados em R (linha 1) e são exclusivos do agente (não se repetem em outros agentes). Se o atributo ($E_{0_{[machine]}}$) corresponder a um evento de R (linha 6), então o agente deverá processar o evento. Neste caso, o agente acionará a função $call\_controller$[1] (linha 8) para proceder com os comandos de controle do recurso

[1] A função $call\_controller$ representa um gatilho para ativar instruções de comando específicas das máquinas no

físico especificado no evento. Ao finalizar o processamento, o evento corrente é excluído do cronograma e o agente notificará para os demais que completou a sua parte da tarefa (linha 10). Este processo se repete no agente sempre que $E_{0_{[machine]}}$ corresponder a um ID de R, até a finalização do cronograma (linha 7).

Algorithm 8 *Synchrony*

```
1:  R ← [Lista dos IDs de máquinas associadas ao agente]
2:  ID ← (ID único de agente)
3:  function PUBLISH_SYNC(old_sync, new_sync)        ▷ Função que atualiza o sync e chama o controle
4:      if (old_sync = sync[s]) then
5:          sync[s] ← new_sync
6:          if (E_0[machine] in R) then               ▷ se a máquina pertence a R
7:              if (|schedule[s]| > 1) then
8:                  call_controller()                 ▷ chama o controle para tratar o evento
9:                  schedule[s].pop(E_0)              ▷ exclui o evento do cronograma de índice s
10:                 publish_sign(1, ID)               ▷ notifica que concluiu o evento
11:             else
12:                 sync[s] ← 0
13:                 schedule.pop(s)
14:                 if (s = 1) then                   ▷ se s = 1, é cronograma urgente
15:                     sync.pop(s)
16:                     s ← 0
17:                     publish_sync(sync[s], sync[s] + 1)   ▷ notificação para retomada da ordem normal
18:                 end if
19:             end if
20:         else                                      ▷ se máquina não pertence a R
21:             flag_lock ← 1                         ▷ bloqueia a atualização de sync
22:             if (|schedule[s]| > 1) then
23:                 call_controller()                 ▷ chama o controle para auxiliar na operação
24:                 schedule[s].pop(E_0)              ▷ exclui o evento do cronograma de índice s
25:             else
26:                 sync[s] ← 0
27:                 schedule.pop(s)
28:                 if (s = 1) then
29:                     sync.pop(s)
30:                     s ← 0
31:                 end if
32:             end if
33:             flag_lock ← 0                         ▷ libera a atualização de sync
34:         end if
35:     end if
36: end function
```

Ao finalizar um cronograma, o índice de controle de sincronismo é reiniciado (linha 12). Se o cronograma finalizado for de ordem urgente (linha 14), o agente notificará os demais com o último valor de $sync$ utilizado para processar o cronograma de ordem normal (linhas 16 e 17). Desta forma, os agentes continuarão processando o cronograma de ordem normal, interrompido pela inserção do cronograma de urgência. Quando um agente não participa diretamente de um evento (linha 20), ele é chamado de coadjuvante; o coadjuvante verifica em seu controle (linha 23) se precisa colaborar com o agente principal no evento atual (p. ex., mover a bandeja de uma mesa giratória). Os demais processos são os mesmos para a exclusão de eventos e para a

ambiente de produção.

finalização de cronogramas, exceto que o agente coadjuvante não envia notificações. Por fim, a *flag* $flag\_lock$ (linhas 21 e 33) garante que o agente não incremente a variável $incr\_sync$ (da *thread* $signaling$) até concluir sua colaboração no evento atual.

Algorithm 9 *Signaling*

```
1:  ID ← (ID exclusivo de agente)
2:  function PUBLISH_SIGN(sign, agent)
3:      checkPriority()                                  ▷ chama o controle de prioridade
4:      if (sign = 1 and agent = ID) then
5:          publish_ntfy(status, 1)
6:          waitSync()                                   ▷ espera sincronização dos agentes p/ a ordem normal
7:          if (E_{0[sync]} = sync[s]) then
8:              publish_sync(sync[s], sync[s])
9:          else
10:             publish_sync(sync[s], sync[s] + 1)
11:         end if
12:     else
13:         waitUnlock()                                 ▷ aguarda liberação de bloqueio do synchrony
14:         publish_ntfy(status, 1)                      ▷ publica mensagem p/ controle do waitSync
15:     end if
16: end function
17: function CHECKPRIORITY()
18:     if (flag_prty = 1) then
19:         if (agent ∈ grp) then
20:             publish_ntfy(priority, 1)
21:         end if
22:         waitPrty()                                   ▷ espera sincronização dos agentes p/ a ordem urgente
23:         sync ← append(0)
24:         s ← 1
25:         flag_incr ← 0
26:         flag_prty ← 0
27:     end if
28: end function
29: function WAITUNLOCK()
30:     while (flag_lock = 1) do
31:         delay(0.1)
32:     end while
33: end function
34: function WAITSYNC()
35:     while (incr_sync < nodesAlive()) do              ▷ verifica o número de agentes ativos p/ a ordem normal
36:         delay(0.1)
37:     end while
38: end function
39: function WAITPRTY()                                  ▷ verifica o número de agentes do grupo p/ a ordem urgente
40:     while (incr_prty < nodesAlive()) do
41:         delay(0.1)
42:     end while
43: end function
```

O código para o tópico *signaling* é apresentado no Algoritmo 9. Sempre que um agente finaliza uma tarefa de um recurso ao qual ele está associado, ele notifica o MAS. Ao receberem qualquer mensagem de sinalização, os agentes verificam se devem mudar suas configurações para operarem com um cronograma de ordem urgente (linha 3). A função $checkPriority$ (linha 17) implementa a mudança de configuração para processamento da ordem de urgência, a partir do momento em que a *flag* $flag\_prty$ se encontrar ativada (linha 18). Caso esteja ativada, cada

agente que integra o grupo responsável pela execução da ordem de urgência (gpr), enviará uma mensagem de notificação para confirmar que está pronto (linhas 19 e 20). Os agentes aguardam até que todos do grupo confirmem estarem prontos (linhas 22, 39 – 43) e, em seguida, a variável s é mudada ($s = 1$), para referenciar o cronograma urgente, e as *flags* $flag\_prty$ e $flag\_incr$ são desativadas (linhas 24 – 26).

A função $publish\_sign$ também possibilita o incremento do $sync$ pelo agente que processa o evento atual, ou um dos agentes que processam eventos paralelos (eventos com o mesmo valor de $E_{0_{[sync]}}$). Para isso, o agente que satisfaz a condição (linha 4), envia uma notificação de "*status*" e aguarda pelos demais agentes (linhas 5, 6, 34 – 38). Os demais agentes (coadjuvantes) também enviam notificações depois que finalizarem as ações do evento atual (linhas 13, 14, 29 – 33). Após a notificação de sincronização dos agentes (linha 6), o agente que satisfez a condição (linha 4) publicará para o MAS o mesmo valor de $sync$ se o evento corrente for paralelo ao executado pelo agente (linhas 7 e 8). Caso contrário, o agente publicará para o MAS um novo valor de $sync$ (linha 10).

O código para o tópico *broadcast* é apresentado no Algoritmo 10. A função $publish\_schedule$ é responsável por receber os cronogramas enviados pelo nó "*dispatcher*" (adaptado dos estudos anteriores) e submetê-lo ao algoritmo FJS-BTC para obtenção do cronograma de produção. Os cronogramas são recebidos evento a evento no tópico *broadcast* e armazenados em um *buffer* temporário (linha 6). Ao final, o *dispatcher* enviará para os agentes um evento especial que indica que o cronograma está completo (linha 2) e, em seguida, o *buffer* é descarregado na variável $schedule$ (linhas 3 e 4); cada agente mantém uma cópia completa do cronograma. Os cronogramas de ordem urgente também são enviados pelo *dispatcher* e recebidos pelos agentes da mesma maneira, exceto que o *dispatcher* também envia uma notificação de prioridade para o último cronograma armazenado na variável $schedule$.

Algorithm 10 *Broadcast*

```
1: function PUBLISH_SCHEDULE(event)
2:     if (event = lastEvent()) then          ▷ verifica se é o último evento recebido
3:         schedule ← append(buffer)          ▷ armazena o cronograma do buffer
4:         buffer ← [ ]                       ▷ libera buffer temporário
5:     else
6:         buffer ← append(event)             ▷ armazena eventos no buffer temporário
7:     end if
8: end function
```

O código associado ao tópico *notification* é apresentado no Algoritmo 11. A função $publish\_ntfy$ é responsável por receber mensagens de notificação e modificar algum parâmetro

de acordo com o tipo de mensagem recebida. Mensagens de notificação do tipo $status$, com valor de mensagem igual a um ($msg = 1$), implicam o incremento de uma variável de sincronismo sempre que algum agente enviar esta notificação. O valor da variável é comparado com o número de agentes ativos do MAS, na função $waitSync$ do algoritmo anterior (ver Alg. 9, linha 35).

Algorithm 11 *Notification*

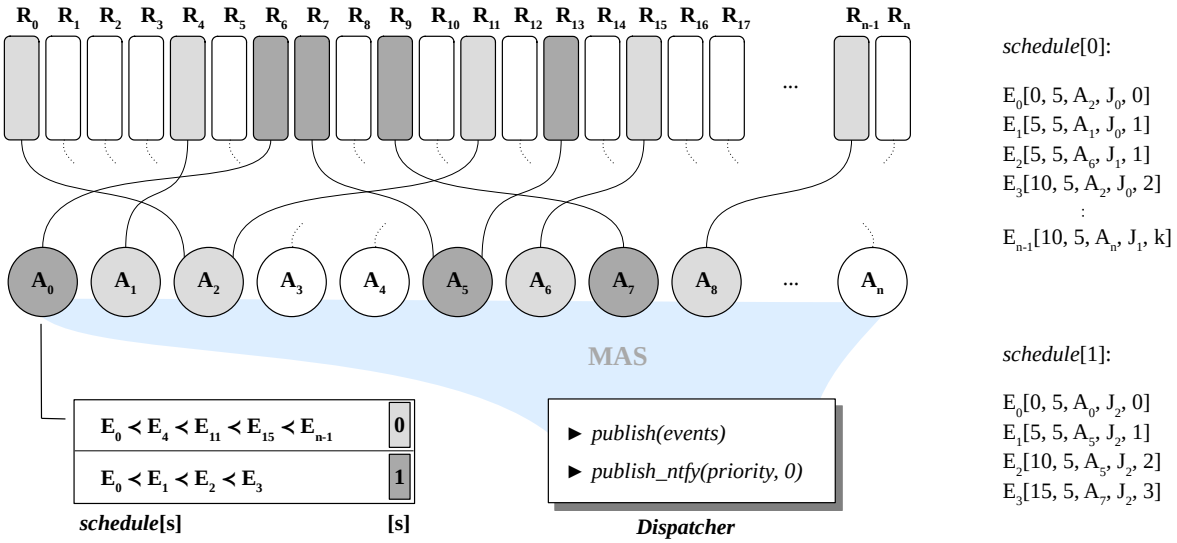
```
1:  function PUBLISH_NTFY(type, msg)                    ▷ função para mensagens de notificação
2:      if (type = status) then
3:          if (msg = 1) then
4:              incr_sync ← incr_sync + 1               ▷ variável de controle para waitSync
5:          end if
6:      else if (type = priority) then
7:          if (msg = 0) then
8:              grp ← nodesPrty(schedule[1])            ▷ obtém grupo resposável pela ordem de urgência
9:              flag_prty ← 1                           ▷ atualiza flag com mensagem do dispatcher
10:         else if (msg = 1) then
11:             incr_prty ← incr_prty + 1               ▷ variável de controle para waitPrty
12:         end if
13:     end if
14: end function
```

A função $publish\_ntfy$ também é responsável por receber notificações do tipo $priority$, relacionadas com as ordens de urgência. Para o valor de mensagem igual a zero ($msg = 0$), exclusivamente enviada pelo nó *dispatcher*, os agentes obtém o grupo de agentes (gpr) responsáveis por executar uma determinada ordem de urgência (linha 8) e habilitam a flag $flag\_prty$ (linha 9). Mensagens com valor igual a um ($msg = 1$), implicam o incremento de outra variável para sincronismo dos agentes (linha 11), antes que iniciem a execução de um cronograma de urgência (ver função $waitPrty$, no Algoritmo 9).

A Figura 23 exemplifica a inserção de uma ordem de urgência, tal que $schedule[1]$ representa o cronograma de ordem urgente ($s = 1$, na cor cinza escuro), enquanto que $schedule[0]$ representa o cronograma de ordem normal ($s = 0$, na cor cinza claro). Cada agente A_i está associado a uma ou mais máquinas ou equipamentos (identificados por R_i) e cada cronograma é composto por uma sequência de eventos E. Neste exemplo não ocorre sobreposição, pois os agentes relacionados com o cronograma de ordem normal são diferentes dos agentes relacionados com o cronograma de ordem urgente. Cada evento E_i indica o agente responsável pela operação e o respectivo job (produto). O cronograma normal possui dois produtos (J_0 e J_1), enquanto que o urgente é possui um (J_2).

O *dispatcher* inicialmente envia o cronograma de ordem normal para o MAS e, ao surgir uma demanda urgente, o cronograma de urgência também é enviado seguido de uma mensagem de notificação de prioridade. A mensagem de prioridade, aos ser recebida pelos agentes, ativa a *flag* $flag\_prty$ (ver Alg. 11, linha 8) e os agentes envolvidos na execução da ordem de urgência

Figura 23 – Exemplo de inserção de ordem de urgência livre de sobreposição.



Fonte: Autoria própria.

(A_0, A_5 e A_7) passam a operar com o novo cronograma. Entretanto, se o cronograma normal e o cronograma urgente contiverem recursos em comum (que se repetem nos cronogramas), é necessário que o MAS identifique o momento adequado de inserção do cronograma de urgência para que não ocorram sobreposições de operações em recursos já ocupados com operações do cronograma normal.

5.2.3 Algoritmo OFTW

O FJS-BTC é apropriado para o controle de consenso proposto para FMSs, pois os agentes são capazes de convergir nos momentos que antecedem as transições de eventos, de acordo com as predefinições de tempo e os índices de controle de sincronismo inseridos nos cronogramas de produção. Como foi discutido, o MAS pode operar com cronogramas preditivos livres de impasses por BTC e suportar a inserção de ordens de urgência. A inserção de uma ordem urgente envolve a seleção de um grupo/*cluster* de agentes para executar o cronograma urgente e, a partir do momento que estão disponíveis para compor o grupo, os agentes devem executar o cronograma independente da situação global do sistema. Isso significa que os agentes do grupo devem chegar a um consenso independente dos demais agentes do MAS, formando assim um consenso por tempo finito pré-definido em grupo/*cluster*.

O Algoritmo 12 trata do tipo de consenso, por tempo finito pré-definido e por grupo/*cluster*, utilizado pelos agentes para a execução dos cronogramas de ordem normal ou

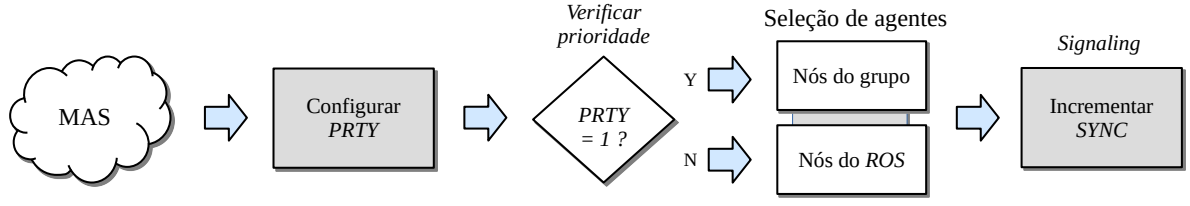
Algorithm 12 *Nodes_Alive & Nodes_Prty*

```
1:  NL ← [<lista nós do MAS>]
2:  function NODESALIVE()
3:      if (flag_prty) then
4:          n ← |grp|                                   ▷ número de nós do grupo
5:      else
6:          state ← ROS.getSystemState()                ▷ lista de parâmetros do ROS
7:          NA ← [ ]
8:          for (s ← state) do
9:              for (l ← s) do
10:                 NA ← extend(l)                      ▷ lista de nós ROS ativos
11:             end for
12:         end for
13:         n ← |NL ∩ NA|                               ▷ filtra os nós do MAS
14:         grp ← [ ]
15:     end if
16:     return(n)
17: end function
18: function NODESPRTY()
19:     grp ← [ ]
20:     B ← schedule[1]                                 ▷ cronograma de ordem urgente
21:     for (b ← 0, |B|) do
22:         agent ← NL[B[b][agent]]                     ▷ obtém o agente do evento
23:         if (agent ∋ grp) then
24:             grp ← append(agent)                     ▷ forma o grupo de agentes
25:         end if
26:     end for
27:     return(grp)
28: end function
```

urgentes. Quando uma mensagem de prioridade é recebida no MAS, é necessário que os agentes formem o grupo responsável pela execução da ordem de urgência. A função $nodesPrty$ é responsável pela formação do grupo (obtido do cronograma), que é armazenado na variável grp (linhas 18 – 27). Se a *flag* $flag\_prty$ estiver ativada (linha 3), a função $nodeAlive$ retornará o número de agentes do grupo (linha 4), utilizado pelo *signaling* para validar o incremento de $sync$ no MAS. Caso contrário, o consenso é obtido com base no número de nós ativos do ROS (linhas 5 – 13).

A Figura 24 resume o processo de seleção dos agentes de acordo com o tipo de prioridade definido para a ordem de produção. A prioridade é alterada quando uma notificação de prioridade é recebida pelo MAS ($PRTY = 1$), ou quando o cronograma urgente é finalizado.

Figura 24 – Seleção do tipo de consenso utilizado pelo MAS.



Fonte: Autoria própria.

Outro ponto fundamental para que um FMS opere com ordens de urgência é garantir que as ordens sejam inseridas livres de impasses por sobreposição, devido a recursos já ocupados com *jobs* do cronograma de produção normal. A inserção de uma ordem de urgência depende do estado atual do FMS e, portanto, não pode ser resolvida por um algoritmo de escalonamento preditivo que gera cronogramas baseados em um estado global inicial do sistema. O algoritmo proposto para esse problema foi chamado de OFTW (*overlay free time window*) e atua combinado ao controle de consenso proposto. O OFTW identifica uma janela de tempo para que a inserção da ordem de urgência ocorra livre de sobreposições, evitando assim *deadlocks* no FMS.

O OFTW é apresentado no Algoritmo 13. A partir de um valor de $sync$, do cronograma normal E, o OFTW verificará se a inserção do cronograma urgente P resultará em algum impasse no FMS. Primeiro, a função $start\_oftw$ utiliza a função $insert\_urgency$ para inserir os eventos do cronograma de urgência no cronograma corrente, a partir do valor de $sync$ utilizado como índice (linhas 46 – 48). O resultado da função é um cronograma modificado S, onde cada evento é formado por $S_i = [start, duration, machine, job, sync, sch]$, tal que $S_{i_{[sch]}}$ indica se os eventos são do cronograma normal (com $sch = 0$) ou do cronograma urgente ($sch = 1$) (linhas 35 – 45).

Em seguida, a função $check\_deadlocks$ verifica nos eventos de ordem normal (linha 14) quais recursos de produção de *buffer* limitado (L) ficarão ocupados e quais resultarão em *deadlocks* por *buffer* nulo (Z), com a inserção do cronograma urgente para o $sync$ corrente. Os recursos ocupados serão registrados em K (linha 17), enquanto que potenciais *deadlocks* serão registrados em N (linha 20) caso a inserção da ordem de urgência provoque impasses. Logo após, a função $check\_deadlocks$ verificará nos eventos urgentes (linha 25 e 26) se há recursos que se sobreporão com a inserção da ordem de urgência para o $sync$ corrente, registrando tais eventos em O (linha 27). Por fim, as constatações da função para o valor de $sync$ serão retornadas (linha 33) e, em caso de impasses, o OFTW testará os efeitos da inserção com os demais valores de $sync$ previstos no cronograma. Se um dado $sync$ (d) não acusar impasses, o algoritmo retornará para os agentes quando eles deverão iniciar a execução da urgência (linhas 50 e 51).

A Figura 25 exemplifica o funcionamento do OFTW. Quando uma notificação de prioridade do *dispatcher* é recebida no MAS, os agentes acionam o algoritmo OFTW para verificar quando a ordem de urgência poderá ser executada livre de impasses por BTC e sobreposição. Considerando que o valor de $sync$ seja igual a 1, no momento que antecede a mudança de eventos do cronograma normal ($schedule[0]$), o OFTW testará a inserção da ordem urgente para o valor de $sync$ imediatamente superior ($sync = 2$) (Figura 25-A). Se o OFTW não identificar impasses,

Algorithm 13 *OFTW*

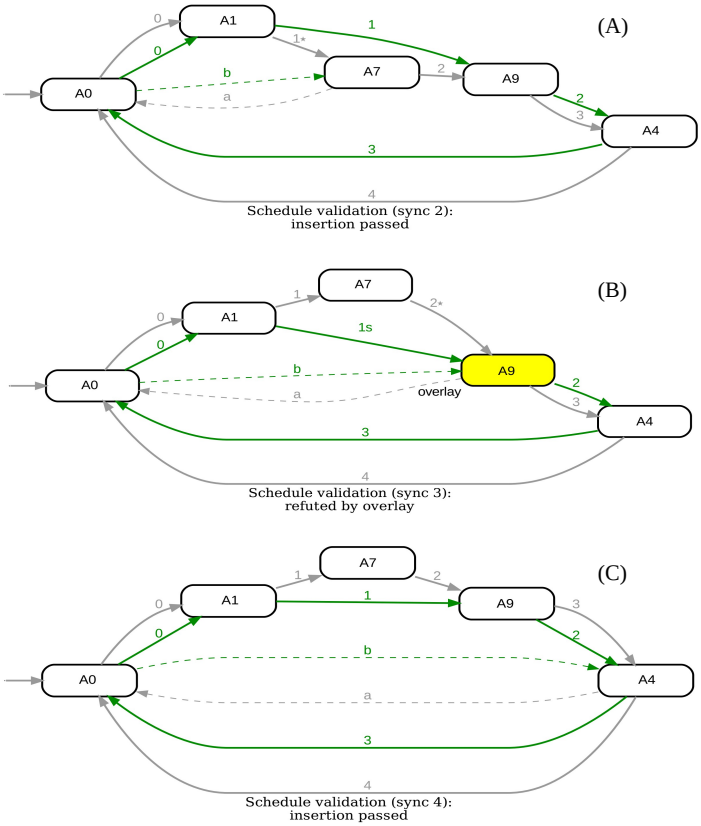
```
1: S, E ← [E_0, E_1, ..., E_n]                                   ▷ cronograma normal
2: P ← [P_0, P_1, ..., P_n]                                      ▷ cronograma urgente
3: L ← [<lista de recursos de buffer limitado>]
4: Z ← [<lista de recursos de buffer nulo>]
5: function CHECK_DEADLOCKS()
6:     O, N, K ← [ ]
7:     for (e ← 0, |S|) do
8:         for (i ← e, |S|) do
9:             src ← S_{e[agent]}
10:            dst ← S_{i+1[agent]}
11:            if (src ∈ K) then
12:                K → remove(src)
13:            end if
14:            if (S_{e[sch]} = 0 and S_{i+1[sch]} = 0) then
15:                if (S_{e[job]} = S_{i+1[job]}) then
16:                    if (dst ∈ L and i <> e) then
17:                        K ← append(dst)                        ▷ recursos com buffer limitado
18:                    else if (dst ∈ Z and i <> e) then
19:                        if (dst ∋ N) then
20:                            N ← append(dst)                    ▷ deadlocks por buffer nulo
21:                        end if
22:                    end if
23:                    break
24:                end if
25:            else if (S_{e[sch]} = 1 and S_{i+1[sch]} = 1) then
26:                if (dst ∈ K and dst ∋ O) then
27:                    O ← append(dst)                            ▷ deadlocks por sobreposição (buffer limitado)
28:                end if
29:                break
30:            end if
31:        end for
32:    end for
33:    return(N, O)
34: end function
35: function INSERT_URGENCY(d)
36:    for (i ← 0, |S|) do
37:        if (S_{i[sync]} > d) then                              ▷ insere cronograma urgente no sync corrente
38:            idx ← S.index(S_i)
39:            for (i ← |P|, 0) do
40:                S ← insert(idx, P_i)
41:            end for
42:            break
43:        end if
44:    end for
45: end function
46: function START_OFTW(sync)
47:    for (d ← sync, |E|) do                                     ▷ sync corrente para o cronograma normal
48:        insert_urgency(d)
49:        N, O ← append(check_deadlocks())
50:        if |N| = 0 and |O| = 0 then
51:            return(d)                                          ▷ retorna que a inserção no (sync) é livre de sobreposições
52:        end if
53:        S ← E
54:    end for
55: end function
```

o grupo de agentes envolvidos com a ordem de urgência inicia a execução do cronograma urgente ($schedule[1]$).

Os agentes executarão os eventos $E_{0_{[A_0]}} \prec E_{1_{[A_1]}} \prec E_{2_{[A_7]}}$ (transições 0 e 1) do cronograma normal (setas na cor cinza), até que o sistema de transporte deixe a peça em A_7

(*buffer* limitado transitório) e retorne descarregado para atender a ordem de urgência (transição a). Os agentes executarão os eventos $E_{0_{[A_0]}} \prec E_{1_{[A_1]}} \prec E_{2_{[A_9]}} \prec E_{3_{[A_4]}} \prec E_{4_{[A_0]}}$ do cronograma urgente (setas na cor verde), até o sistema de transporte retornar descarregado para A_7 (transição b). Por fim, os eventos $E_{2_{[A_7]}} \prec E_{3_{[A_9]}} \prec E_{4_{[A_4]}} \prec E_{5_{[A_0]}}$ (transições 2–5) que restavam do cronograma normal serão executados.

Figura 25 – Exemplo de funcionamento do algoritmo OFTW.



schedule[0]:

E_0[0, 5, A_0, J_0, 0]
E_1[5, 5, A_1, J_0, **1**] **sync**
E_2[10, 5, A_7, J_0, 2]
E_3[15, 5, A_9, J_0, 3]
E_4[20, 5, A_4, J_0, 4]
E_5[25, 5, A_0, J_0, 5]

schedule[1]:

E_0[0, 5, A_0, J_0, 0]
E_1[5, 5, A_1, J_0, 1]
E_2[10, 5, A_9, J_0, 2]
E_3[15, 5, A_4, J_0, 3]
E_4[20, 5, A_0, J_0, 4]

Fonte: Autoria própria.

Se o $sync$ do momento fosse igual a 2, o algoritmo OFTW testaria a inserção da ordem de urgência para o valor de $sync$ igual a 3 (Figura 25-B). De acordo com o diagrama de estados, os agentes executariam os eventos $E_{0_{[A_0]}} \prec E_{1_{[A_1]}} \prec E_{2_{[A_7]}} \prec E_{3_{[A_9]}}$ do cronograma original (transições 0–2), até o sistema de transporte deixar a peça em A_9 e retornar descarregado (transição a). Em seguida, os agentes executariam os eventos $E_{0_{[A_0]}} \prec E_{1_{[A_1]}}$, mas não poderiam depositar a peça no recurso associado a A_9 (evento $E_{2_{[A_9]}}$) por estar ocupado com o *job* (peça) do cronograma original. Logo, devido ao impasse por sobreposição, o OFTW testaria a inserção da ordem de urgência para o próximo valor de $sync$ ($sync = 4$) (Figura 25-C). Primeiro os agentes executariam os eventos $E_{0_{[A_0]}} \prec E_{1_{[A_1]}} \prec E_{2_{[A_7]}} \prec E_{3_{[A_9]}} \prec E_{4_{[A_4]}}$ do cronograma normal, até o sistema de transporte depositar a peça em A_4 e retornar vazio (transição a).

Em seguida, o grupo de agentes responsável pelo cronograma urgente executaria os eventos $E_{0_{[A_0]}} \prec E_{1_{[A_1]}} \prec E_{2_{[A_9]}} \prec E_{3_{[A_4]}} \prec E_{4_{[A_0]}}$, até o sistema de transporte retornar descarregado (transição b). Por fim, o MAS executaria os eventos $E_{4_{[A_4]}} \prec E_{5_{[A_0]}}$ para finalizar o cronograma normal. Portanto, considerando que a inserção do cronograma urgente é inviável na transição do $sync$ 2 para o $sync$ 3, o OFTW retornará para o MAS que a inserção deverá ocorrer na transição do $sync$ 3 para o $sync$ 4.

5.3 RESULTADOS E DISCUSSÃO

Esta seção descreve os experimentos realizados com os algoritmos FJS-BTC e OFTW, para a avaliação dos métodos de obtenção de cronogramas, reajustamentos e validação das ordens de urgência. Os resultados experimentais estão divididos em duas subseções. Um computador Intel Core i5-8250U 1,60 GHz com 8 GB de RAM foi utilizado para obter os tempos de processamento com os algoritmos. Os agentes foram configurados como processos do sistema operacional e classificados como *buffer* nulo ou limitado para representar um ambiente de produção com BTCs. Primeiramente, foram avaliados os métodos utilizados pelo FJS-BTC para a obtenção dos cronogramas de produção. Em seguida, resultados sobre o reajustamento dos cronogramas preditivos são apresentados. Por fim, são apresentados os resultados do algoritmo OFTW para a inserção de ordens de urgência e uma comparação entre um algoritmo de escalonamento convencional e o algoritmo FJS-BTC. Um material complementar com os cronogramas, *logs* de execução dos algoritmos, diagramas de Gantt e de estado estão disponíveis online[2].

5.3.1 Experimentos com o algoritmo FJS-BTC

Este experimento compara os métodos do algoritmo FJS-BTC na obtenção de composições para os modelos de produção. Em seguida, são apresentados dados referentes ao reajustamento de cronogramas livres de impasses por BTC utilizando o filtro de três estágios. A Tabela 15 apresenta resultados para cinco cronogramas de produção gerados a partir de matrizes quadradas de 3×3 até 7×7 ($m \times n$). Cada linha da matriz (m) representa a definição de um produto e cada coluna (n) representa o número de operações por produto (*job*).

Para cada cronograma, é apresentada a sequência de distribuição selecionada pelos métodos, bem como o número de permutações (*Perm.*), eventos, *syncs*, transições, *Length*

[2] Material complementar: https://github.com/alexlds77/fjs-btc

Tabela 15 – Resultados do FJS-BTC com matrizes quadradas.

Sch.	Modelo (m×n)	Método	Distrib.	Perm.	Eventos	*Sync*	Trans.	*Length* (s)	*Time* (ms)
1	3 x 3	*Easy-Perm*	[0, 2, 1]	6	9	4	13	25	0.56
		Circular-Perm	[0, 2, 1]	2	9	4	13	25	0.19
		Easy-Shift	[0, 1, 2]	3	9	5	12	30	0.38
		Cyclic-Shift	[0, 2, 1]	6	9	4	13	25	0.52
		Random-Array	[0, 2, 1]	6	9	4	13	25	0.58
2	4 x 4	*Easy-Perm*	[3, 0, 2, 1]	24	16	7	25	45	3.66
		Circular-Perm	[0, 3, 2, 1]	6	16	7	25	45	0.90
		Easy-Shift	[3, 0, 1, 2]	4	16	8	23	50	0.76
		Cyclic-Shift	[3, 0, 2, 1]	12	16	7	25	45	1.83
		Random-Array	[0, 3, 2, 1]	12	16	7	25	45	1.78
3	5 x 5	*Easy-Perm*	[2, 4, 1, 3, 0]	120	25	9	41	59	26.09
		Circular-Perm	[0, 4, 1, 3, 2]	24	25	9	41	59	4.97
		Easy-Shift	[4, 0, 1, 2, 3]	5	25	9	43	61	1.10
		Cyclic-Shift	[4, 0, 1, 2, 3]	20	25	9	43	61	4.18
		Random-Array	[0, 4, 1, 3, 2]	22	25	9	41	59	4.73
4	6 x 6	*Easy-Perm*	[3, 0, 1, 2, 5, 4]	720	36	12	65	76	195.29
		Circular-Perm	[0, 3, 5, 2, 4, 1]	120	36	12	63	77	31.02
		Easy-Shift	[5, 0, 1, 2, 3, 4]	6	36	12	64	78	1.63
		Cyclic-Shift	[5, 0, 1, 2, 3, 4]	30	36	12	64	78	7.94
		Random-Array	[2, 5, 3, 1, 4, 0]	35	36	12	64	77	9.79
5	7 x 7	*Easy-Perm*	[0, 6, 1, 5, 3, 4, 2]	5040	49	14	89	91	1793.56
		Circular-Perm	[0, 6, 1, 5, 3, 4, 2]	720	49	14	89	91	252.86
		Easy-Shift	[0, 1, 2, 3, 4, 5, 6]	7	49	14	88	94	2.62
		Cyclic-Shift	[0, 1, 2, 3, 6, 4, 5]	42	49	14	89	94	14.86
		Random-Array	[1, 3, 6, 4, 2, 5, 0]	48	49	14	88	94	17.42

Fonte: Autoria própria.

(*makespan* teórico, em segundos) e tempo de processamento (*Time*, em milissegundos). A sequência de distribuição (*Distrib.*) refere-se à configuração das linhas nas matrizes modelo. Por exemplo, a sequência $[0, 2, 1]$ representa a matriz 3×3 com menor valor de *length* obtido pelo método de permutação simples, onde o produto J_0 está na primeira linha da matriz, o produto J_2 na segunda linha e o produto J_1 na terceira.

Considerando que o método de permutação simples gera todas as distribuições possíveis de matriz modelo obtendo o menor *length*, pode-se comparar que o valor de *length* para os outros métodos é maior ou igual ao obtido com a permutação simples. Da mesma forma, o número de *syncs* obtido com os outros métodos também é maior ou igual ao obtido com a permutação simples. Porém, quanto maior for o número de *syncs*, maior é a serialização de eventos no cronograma (aumentando o valor de *length*). Em comparação aos outros métodos, o tempo de processamento para a permutação simples é bem maior, se tornando inviável para

calcular cronogramas grandes devido à quantidade de permutações efetuadas. Entretanto, os outros métodos nem sempre podem garantir o menor valor de *length*, como ocorre no cronograma 4 (*Sch*. 4). Porém, um critério que pode ser adotado para a escolha do cronograma e que está diretamente relacionado às BTCs, é o número de transições. Quanto menor for o número de transições, menos movimentos efetuados pelo sistema de transporte serão necessários. Entretanto, seria preciso considerar as distâncias percorridas para avaliar se o cronograma é melhor.

Tabela 16 – Resultados do FJS-BTC com cronogramas reajustados.

Sch.	Eventos por *job*	*Syncs*				Transições				*Length* (s)			
		STD	SRF	PRF	OVF	STD	SRF	PRF	OVF	STD	SRF	PRF	OVF
1	$J_0(6), J_1(6), J_2(5), J_3(6), J_4(5)$	10	12	12	13	47	40	40	38	64	74	74	82
2	$J_0(8), J_1(7), J_2(5), J_3(6), J_4(5), J_5(7)$	14	18	18	19	65	54	54	53	93	113	113	120
3	$J_0(5), J_1(5), J_2(4), J_3(5), J_4(4), J_5(5), J_6(6), J_7(5)$	15	16	16	16	66	53	53	53	92	97	97	97
4	$J_0(5), J_1(6), J_2(6), J_3(6), J_4(5)$	9	13	13	15	48	38	38	36	60	83	83	95
5	$J_0(7), J_1(5), J_2(6), J_3(5), J_4(7), J_5(5), J_6(6)$	13	17	17	18	72	59	59	58	88	111	111	114

Fonte: Autoria própria.

A Tabela 16 apresenta resultados para 5 cronogramas FJS-BTC livres de impasses após reajustamento com o filtro de três estágios. O número de produtos por cronograma varia entre 5 e 8, e o número de operações por produto (J_i), para cada cronograma, é apresentado entre parênteses. Por exemplo, o cronograma 1 é composto por $J_0(6), J_1(6), J_2(5), J_3(6), J_4(5)$; significa que há 6 operações de fabricação para o produto J_0, 6 operações para J_1, 5 operações para J_2, 6 operações para J_3 e 5 operações para J_4, totalizando 28 eventos no cronograma. O número de *syncs*, transições e *length* também são apresentados para cronogramas sem reajustamento (STD), reajustados com serialização e podagem (SRF + PRF), ou reajustados por sobreposição (OVF). O reajustamento por sobreposição também inclui os reajustes de serialização e de podagem.

Em comparação com os cronogramas sem reajustamento, o número de *syncs* para os cronogramas reajustados pelos filtros é maior, devido à redução de eventos executados paralelamente para resolver problemas de BTC e devido ao deslocamento de eventos para problemas de sobreposição. Consequentemente, o valor de *length* também aumenta em relação ao *length* dos cronogramas sem reajustamento, mas os cronogramas reajustados podem ser executados livres de impasses. Do contrário, empregar cronogramas sem reajustamento em FMSs com restrição de *buffer* e de transporte é impraticável, pois o sistema de transporte compartilhado não está disponível prontamente.

5.3.2 Experimentos com o algoritmo OFTW

Este experimento primeiro apresenta resultados do algoritmo OFTW para a inserção de ordens de urgência em cinco cronogramas de produção de ordem normal. Em seguida, é apresentada uma comparação entre um algoritmo de escalonamento convencional e o algoritmo FJS-BTC proposto. A Tabela 17 apresenta resultados de cronogramas formados por três produtos distintos e os respectivos agentes envolvidos na fabricação de cada produto, incluindo o número de eventos de cada cronograma, *syncs*, transições e *length* (*makespan* teórico, em segundos).

Tabela 17 – Resultados do OFTW para a validação de ordens urgentes.

Sch.	Ordem	Agentes	Eventos	*Syncs*	Trans.	*Length*	Avaliação			
							Aceitar	Rejeitar	Z	L
1	Normal	A0, A1, A7, A8, A4, A0; A0, A4, A5, A6, A3, A14, A0; A0, A1, A9, A14, A0;	18	12	20	65	0; 8–12	1–7	1(1); 2(4); 3(5); 4(6); 5(3); 6(1)	7(8)
	Urgente	A0, A1, A8, A14, A0;	5	4	4	25				
2	Normal	A0, A1, A8, A9, A4, A0; A0, A4, A5, A6, A3, A14, A0; A0, A1, A9, A14, A0;	18	12	20	65	0; 9–12	1–8	1(1); 2(4); 3(5); 4(6); 5(3); 6(1)	7(9); 8(9)
	Urgente	A0, A1, A9, A14, A0;	5	4	4	25				
3	Normal	A0, A1, A7, A11, A4, A0; A0, A4, A9, A8, A3, A14, A0; A0, A1, A12, A2, A14, A0;	19	10	25	55	0; 4–6; 9; 10	1–3; 7; 8	1(1); 2(4); 3(1); 7(3); 8(2)	2(7); 3(7)
	Urgente	A0, A1, A7, A3, A14, A0;	6	5	5	30				
4	Normal	A0, A1, A11, A12, A4, A0; A0, A4, A8, A9, A3, A14, A0; A0, A1, A9, A2, A14, A0;	19	12	23	65	0; 10–12	1–9	1(1); 2(4); 3(1); 5(2); 9(3)	4(9); 6(9); 7(9) 8(9)
	Urgente	A0, A1, A9, A2, A4, A0;	6	5	5	30				
5	Normal	A0, A1, A7, A13, A9, A4, A0; A0, A4, A10, A9, A3, A2, A14, A0; A0, A1, A5, A6, A14, A0;	21	12	26	65	0; 6; 9–12	1–5; 7; 8	1(4); 2(1); 3(5); 4(6); 5(1); 7(3); 8(2)	-
	Urgente	A0, A1, A3, A8, A12, A2, A14, A0;	8	7	7	40				

Fonte: Autoria própria.

A Tabela 17 ainda destaca a avaliação das inserções para cada valor de $sync$ (em *Syncs*) dos cronogramas normais, com a indicação dos valores em que a inserção é bem-sucedida (*Aceitar*) ou não (*Rejeitar*). Os impasses por *buffer* nulo (Z) e por *buffer* limitado (L) (sobreposição), com a identificação dos valores de $sync$ e respectivos agentes (entre parênteses), também são apresentados. De acordo com a tabela, é possível notar que o número de *syncs* que permitem a inserção do cronograma urgente sem que ocorram impasses é, em geral, menor que o número de *syncs* que não permitem. Entretanto, o momento de inserção ideal dependerá de quando a ordem de prioridade será enviada para o MAS, que é algo impossível de prever.

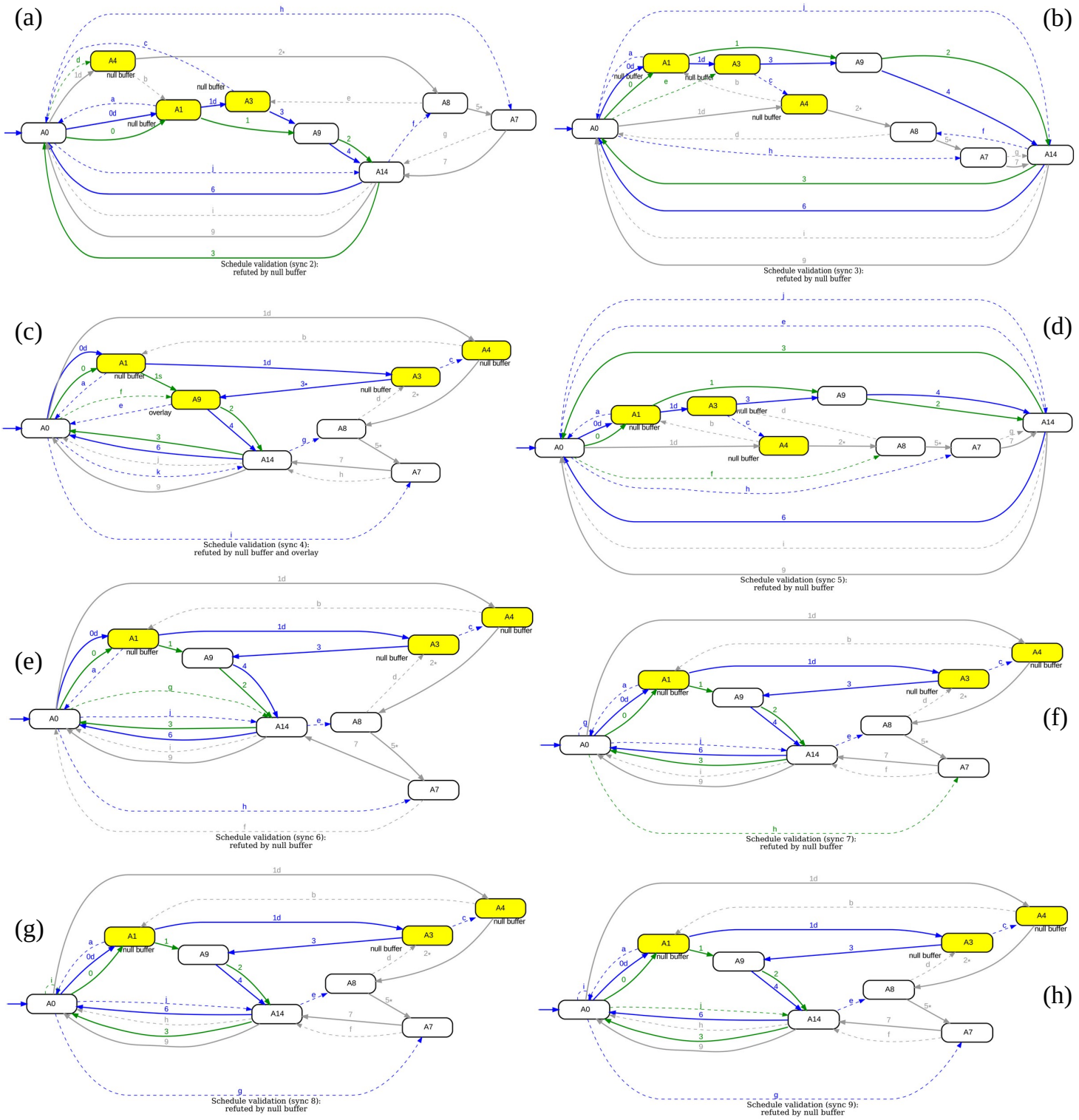
Por fim, é apresentada uma comparação entre um algoritmo de escalonamento convencional (do pacote *OR-Tools*, da Google) e o algoritmo FJS-BTC proposto. O algoritmo OFTW é utilizado para retornar, se possível, os momentos adequados de inserção da ordem

de urgência no cronograma de ordem normal. A comparação considera um cronograma de ordem normal formado por dois produtos distintos e um cronograma de ordem urgente formado por um outro produto. No cronograma normal, o primeiro produto é definido pela sequência $A_0 \prec A_1 \prec A_2 \prec A_3 \prec A_2 \prec A_{14} \prec A_0$, enquanto que o segundo é definido pela sequência $A_0 \prec A_4 \prec A_2 \prec A_3 \prec A_4 \prec A_0$. No cronograma urgente, o produto é definido pela sequência de produção $A_0 \prec A_4 \prec A_3 \prec A_{14} \prec A_0$. Como as sequências possuem agentes em comum, significa que os recursos associados aos agentes são compartilhados para a fabricação dos diferentes tipos de produtos, situação que sujeita o FMS a impasses.

A Figura 26 apresenta os diagramas de estados para o algoritmo de escalonamento convencional. Nos diagramas estão destacados os *deadlocks* (na cor amarela), com a indicação de *buffer* nulo ou sobreposição, para cada *sync* do cronograma. As transições são representadas por setas contínuas, quando o transporte se movimenta carregado, ou setas tracejadas quando se movimenta descarregado (vazio). As transições na cor verde estão relacionadas com o cronograma de urgente, enquanto que as demais (cinza e azul) estão relacionadas com o cronograma de ordem normal. Na Figura 26, os diagramas A – H apresentam *deadlocks* por *buffer* nulo nos recursos associados aos agentes A_1, A_3 e A_4. Além disso, o diagrama C também apresenta *deadlock* por sobreposição no recurso associado ao agente A_9. Os valores de *sync* iguais a 0, 1 e 10 também apresentam *deadlocks*, mas os diagramas foram omitidos da figura para melhorar a apresentação. Portanto, a ordem de urgência não pode ser inserida em nenhum momento no sistema porque o cronograma gerado pelo algoritmo de escalonamento convencional não considera BTCs.

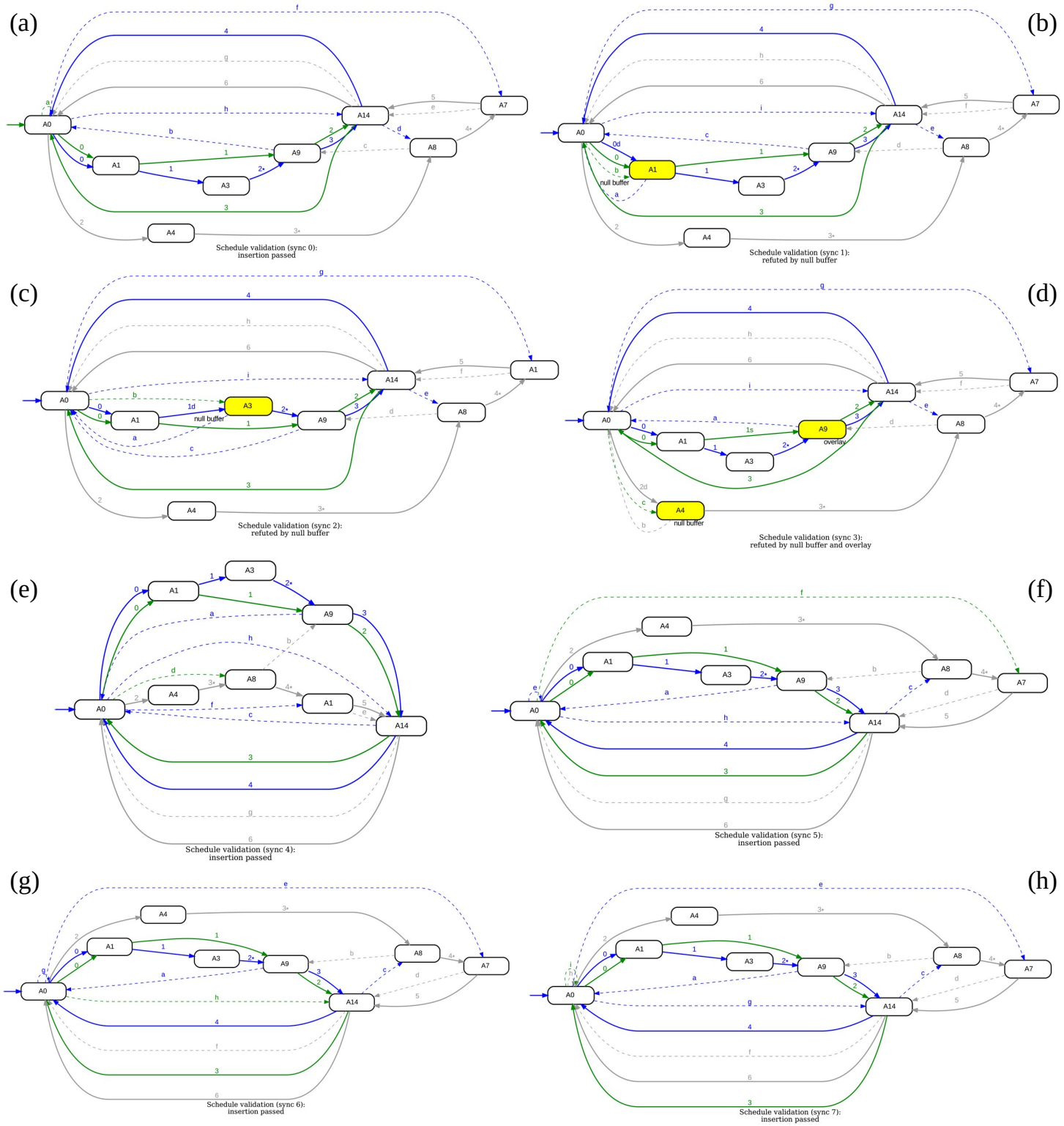
A Figura 27 apresenta os diagramas de estados do OFTW para os mesmos cronogramas (normal e urgente), porém gerados e reajustados pelo algoritmo FJS-BTC. Ao contrário do algoritmo de escalonamento convencional, os diagramas A, E – H, mostram que é possível inserir o cronograma urgente em 5 oportunidades, ou seja, nos $syncs$ 0, 4 – 7. Entretanto, nos diagramas B – D a inserção da ordem de urgência implicará em *deadlocks* por *buffer* nulo e por sobreposição. Mas é importante destacar que o OFTW garante a inserção de ordens de urgência em FMSs com produção controlada por ordem e utilizando cronogramas preditivos, sem a necessidade de utilizar métodos de reescalonamento computacionalmente caros em um sistema de produção em operação. Basta avaliar se é mais vantajoso para a indústria que o reescalonamento seja instituído, ou se é mais vantajoso aguardar pelos momentos certos de inserção das urgências.

Figura 26 – Diagramas de estado para o algoritmo de escalonamento convencional.



Fonte: Autoria própria.

Figura 27 – Diagramas de estado para o algoritmo FJS-BTC.



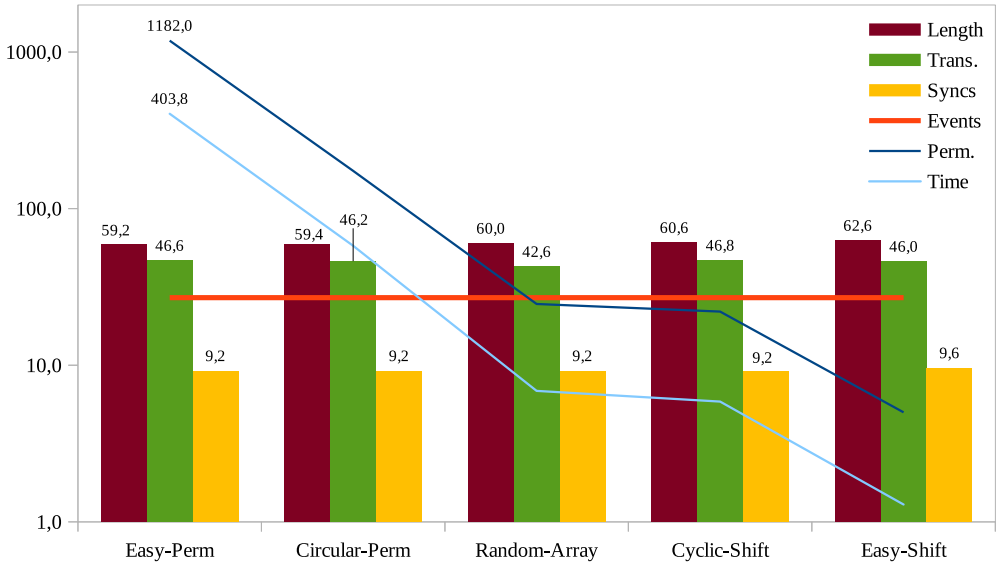
Fonte: Autoria própria.

5.4 AVALIAÇÃO GERAL

Esta seção apresenta a avaliação geral das propostas deste estudo e a consolidação dos resultados obtidos com os experimentos. A Figura 28 apresenta o gráfico com a média das

métricas utilizadas pelo algoritmo FJS-BTC para os dados do primeiro experimento (Tabela 15). Os resultados comprovaram que o *Easy-Perm* obteve os melhores valores de *length*, apesar do tempo de processamento relativamente elevado. Por outro lado, o *Circular-Perm* obteve valores de *Length* muito próximos aos do *Easy-Perm*, e com um tempo de processamento 85% menor (idem para o número de permutações). O *Random-Array* também foi capaz de obter valores aceitáveis de *length*, variando cerca de 1.35% dos valores de *length* obtidos pelo *Easy-Perm*. Além disso, o somatório dos tempos médios de processamento do *Circular-Perm*, *Random-Array*, *Cyclic-Shift* e *Easy-Shift* é 82,17% menor que o tempo do *Easy-Perm*. Isso significa que estes métodos podem ser combinados para a obtenção de cronogramas quase ótimos, e com desempenho superior ao *Easy-Perm*.

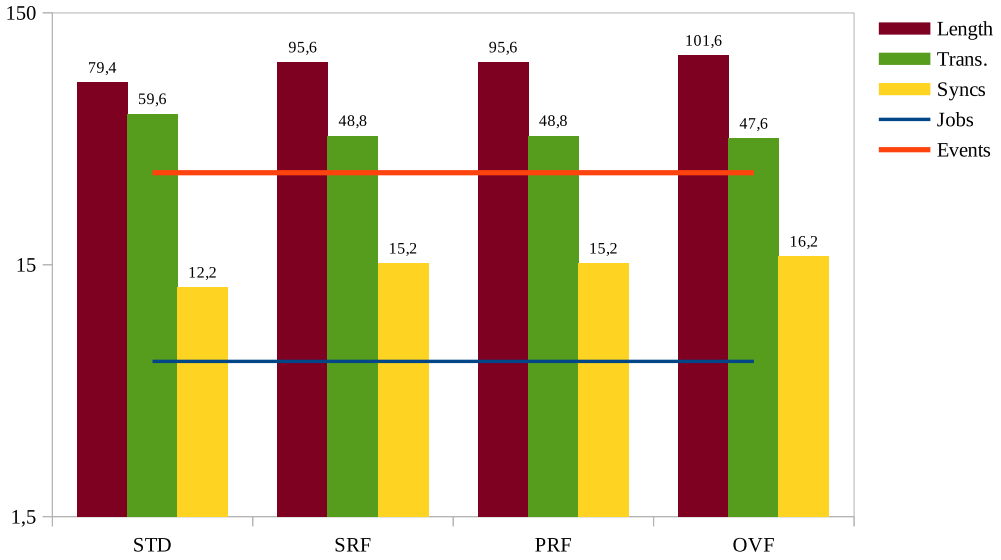
Figura 28 – Médias das métricas do algoritmo FJS-BTC.



Fonte: Autoria própria.

A Figura 29 apresenta um gráfico de comparação entre o algoritmo de escalonamento convencional (STD) e o FJS-BTC (filtros SRF, PRF e OVF), utilizando as médias dos dados da segunda tabela do primeiro experimento (Tabela 16). De acordo com o gráfico, o número de transições reduziu 20,13% com o OVF (que já inclui SRF e PRF), em comparação ao algoritmo convencional (STD). Esta redução é devido ao reajustamento feito pelo FJS-BTC para a adequação dos cronogramas. Entretanto, no STD este percentual representa *deadlocks* e comprova que cronogramas gerados para ambientes sujeitos a BTCs são falhos porque permitem transições que consideram a imediata disponibilidade do sistema de transporte compartilhado. O reajustamento eleva em 21,85% o valor de *length*, mas garante cronogramas livres de impasses por BTCs sem a aplicação de reescalonamento.

Figura 29 – FJS-BTC vs. Escalonamento preditivo convencional.



Fonte: Autoria própria.

O algoritmo FJS-BTC é capaz de gerar cronogramas preditivos com valores de *makespan* apropriados para FMSs com produção controlada por ordem, em ambientes *multi-product multi-machine* sujeitos à BTCs. A obtenção de *makespan* por valores máximos permite ao FMS maior previsibilidade nos processos de produção, que pode ser difícil de obter com algoritmos de escalonamento convencionais. A previsibilidade é um dos pontos mais importantes da indústria moderna para manter uma gestão eficaz e dimensionar com maior precisão a sua produção. O controle também é fundamental para o sucesso de um FMS; entende-se que escalonamento e controle são indissociáveis para o desenvolvimento de soluções com MAS em problemas de produção. No controle o aproveitamento do consenso multiagente para a inserção das ordens de urgência em períodos pré-definidos também favorece a previsibilidade na produção, que pode não ser obtida com a mesma facilidade em abordagens que visam reescalonamento.

A Tabela 18 apresenta dados sobre a inserção das ordens de urgência relacionadas com os cronogramas FJS-BTC do segundo experimento (Tabela 17). Os dados mostram que o algoritimo OFTW validou (“*Aprovados*”) a inserção em 42.86% dos *syncs* relacionados com os cronogramas regulares (corresponde a 27 *syncs*, em “*A*”). Em 57,14% dos *syncs* a inserção não foi validada (“*Reprovados*”) devido a problemas de BTC, sendo 42.86% por *buffer* nulo (“*Z*”), 11.11% por *buffer* limitado (“*L*”) e 3.17% por ambos os impasses (por *buffer* nulo e *buffer* limitado). Os dados consolidam os resultados dos experimentos e comprovam que a abordagem com FJS-BTC, MAS e OFTW é eficiente para uma produção livre de impasses, possibilitando flexibilidade e previsibilidade em FMS sujeitos à BTCs.

Tabela 18 – Percentual de validação do algoritmo OFTW para as ordens de urgência.

Sch.	Eventos	*Syncs* por cronograma		Trans.	*Length* (s)	Avaliação de *syncs*			Validação (%)	
		Regular	Urgente			A	Z	L	Aprovados	Reprovados
1	23	13	5	24	90	6	6	1	9.52	11.11
2	23	13	5	24	90	5	6	2	7.94	12.70
3	25	11	6	30	85	6	5	2	9.52	7.94
4	25	13	6	28	95	4	5	4	6.35	14.29
5	29	13	8	33	105	6	7	0	9.52	11.11
Total	**125**	**63**	**30**	**139**	**465**	**27**	**29**	**9**	**42.86**	**57.14**

Fonte: Autoria própria.

A relação entre FJS-BTC e OFTW poderá ser melhor estudada em trabalhos futuros para tornar a solução final mais robusta. Na prática, outras matrizes geradas pelo FJS-BTC apresentam a mesma dimensão e valor de *makespan* que a matriz ótima, porém aquela que assume no primeiro momento o papel de menor em termos de dimensão e de *makespan* é a que permanece até o final como ótima. Entretanto, se a matriz ótima for classificada no reajustamento como inadequada devido à impasses, ou aumentar muito a sua dimensão com os processos de serialização e sobreposição, outras matrizes candidatas à ótimas poderiam ser avaliadas para substituir a matriz ótima inicial. Uma outra possibilidade de melhoria para o presente estudo está relacionada com o sincronismo, que ocorre somente quando a operação mais demorada (com o *sync* corrente) dos agentes em execução é finalizada. Entretanto, a espera pelo término da operação mais demorada ocasiona períodos de indisponibilidade de uso das máquinas que finalizaram operações mais curtas. Apesar de ser uma característica prevista no FJS-BTC, devido à espera pela disponibilidade do sistema de transporte em recursos de *buffer* nulo, o incremento de sincronismo poderia ser forçado via controle nos casos em que as operações independem do transporte (por exemplo, em recursos de *buffer* limitado).

Tabela 19 – Complexidade de tempo dos algoritmos propostos.

	Melhor caso	Pior caso
Easy Perm.	$\Omega(n^2 + 3n + 3)$	$O(n^2 + 3n + 3)$
Circular Perm.	$\Omega(n^2 + 6n + 5)$	$O(n^2 + 6n + 5)$
Easy Shift	$\Omega(n^2 + n + 3)$	$O(n^2 + n + 3)$
Cyclic Shift	$\Omega(n + 3)$	$O(n^2 + 3n + 5)$
Random Array	$\Omega(4n + 8)$	$O(5n + 8)$
FJS-BTC	$\Omega(4n^2 + 13n + 36)$	$O(6n^2 + 31n + 58)$
OFTW	$\Omega(n^2 + 15n + 29)$	$O(n^2 + 25n + 38)$

Fonte: Autoria própria.

A Tabela 19 apresenta dados sobre a complexidade de tempo dos algoritmos propostos. As notações Ω e O foram utilizadas para expressar o limite inferior (melhor caso) e limite superior

(pior caso) dos algoritmos. A complexidade calculada para os métodos de obtenção dos modelos de produção utilizados pelo FJS-BTC são apresentadas à parte. Os processos de adequação dos modelos, verificação de sobreposições, deslocamentos, cálculo da matriz ótima e geração do cronograma final se referem à complexidade do FJS-BTC em processos comuns para os cinco métodos de obtenção de modelos. Por exemplo, a complexidade do algoritmo FJS-BTC para o método *Easy Perm* é o valor de complexidade do algoritmo somado com o valor de complexidade do método. Por fim, a complexidade de tempo do algoritmo OFTW também é apresentada e refere-se a um único agente, pois os códigos de identificação de impasses e de consenso são idênticos no MAS. O cálculo de complexidade de tempo dos algoritmos de consenso e de reajustamento não foram incluídos na tabela porque se assemelham aos apresentados no estudo anterior (Capítulo 4).

5.5 CONCLUSÕES DO CAPÍTULO

Este estudo apresentou um novo algoritmo de escalonamento preditivo, chamado FJS-BTC, e um algoritmo para validar a inserção de ordens de urgência em FMSs. O FJS-BTC é um algoritmo para problemas de escalonamento com restrições por BTC, capaz de gerar cronogramas de produção livres de *deadlocks* por insuficiência de *buffer* e por sobreposição. O FJS-BTC é especialmente adequado para FMSs que operam com ordens de produção em ambientes limitados de *buffer* e de transporte, pois fornece valores de *makespan* mais realistas para gerenciamento e previsibilidade da produção. Cinco métodos de obtenção de modelos de produção podem ser utilizados pelo FJS-BTC para gerar cronogramas preditivos. O método de permutação simples pode garantir cronogramas ótimos, entretanto é lento computacionalmente dependendo do tamanho da entrada fornecida. Uma adaptação que poderia ser experimentada em trabalhos futuros é considerar o número de transições de eventos na seleção da matriz ótima. Outros métodos mais rápidos também podem obter cronogramas aceitáveis, mas não garantem cronogramas ótimos. A combinação destes métodos poderia melhorar a solução, um caminho é o uso de um mecanismo de poda similar ao *Branch-and-Bound*.

Experimentos mostraram que o FJS-BTC, combinado a consenso multiagente por tempo finito pré-definido e por grupo/*cluster*, pode garantir a inserção de ordens de urgência em FMSs sem a necessidade de reescalonamento durante a produção (*runtime*). O controle de consenso fornece sincronismo, possibilitando que métodos inteligentes de apoio à decisão sejam executados pelos agentes durante os períodos de convergência do MAS. Aproveitando essa

característica, desenvolveu-se um algoritmo chamado OFTW, que utiliza as janelas de tempo de convergência para validar a inserção das ordens de urgência no sistema. O OFTW observa potenciais impasses por BTC e por sobreposição, que podem ocorrem com a inserção das ordens urgentes em recursos de produção compartilhados. A abordagem é relevante porque combina um novo algoritmo preditivo com suporte à BTCs e consenso multiagente, para a execução de cronogramas normais e urgentes (livres de impasses) em FMSs controlados por ordem. A facilidade dos agentes em obterem informações sobre o estado do sistema, também possibilita que a abordagem seja combinada às outras soluções.

6 CONCLUSÕES

Esta tese desenvolveu estudos na temática de escalonamento e controle da produção, com foco em descentralização, flexibilidade, prevenção de impasses e previsibilidade da produção, com o aproveitamento do consenso multiagente. O primeiro estudo propôs um MAS distribuído e abordagens de controle de impasses para experimentação em uma fábrica virtual. Um controle de consenso sem líderes foi utilizado para a coordenação dos agentes em operações de produção definidas por um algoritmo de escalonamento de *job-shop* flexível (FJS). Dois algoritmos de controle adicionais, chamados *Classifier* e *Combiner*, foram desenvolvidos e comparados. O algoritmo *Classifier* se baseou na teoria da evidência de Dempster-Shafer, permitindo que os agentes escolhessem o melhor caso de controle livre de impasses para conduzir a produção. O algoritmo *Combiner* foi implementado com uma tabela de decisão que garantiu o roteamento de produtos na fábrica. Um algoritmo chamado *Preset* também foi associado ao *Combiner* para ajustar os cronogramas e evitar impasses no sistema. Os experimentos mostraram que a solução *Combiner* + *Preset* foi mais eficiente para o conjunto de cronogramas testados. Porém, o *Classifier* (Dempster-Shafer) pode fornecer uma estrutura mais adequada para os agentes lidarem com problemas que envolvam tomadas de decisões. Os objetivos relacionados com o escalonamento e a descentralização do controle foram atingidos e dois artigos científicos foram produzidos neste estudo.

O segundo estudo modificou uma plataforma física de experimentação para operar como um FMS com produção controlada por ordem. Um MAS distribuído e de arquitetura em camadas foi implementado para substituir o controle centralizado de ciclo de produção fixo da plataforma física original. A nova plataforma de experimentação de FMS pôde suportar produção flexível de acordo com cronogramas gerados por um algoritmo de FJS. Três filtros de reajustamento de cronogramas preditivos foram desenvolvidos para garantir flexibilidade e evitar impasses no sistema. O primeiro filtro foi capaz de identificar eventos associados a recursos de *buffer* nulo e serializar esses eventos para que não ocorressem paralelamente nos cronogramas. O segundo filtro pôde identificar eventos com recursos de *buffer* nulo organizados contiguamente a eventos de *buffer* limitado, priorizando estes eventos e dispondo-os em paralelo para otimizar os cronogramas. O terceiro filtro foi capaz de identificar sobreposições e evitá-las trocando a posição dos eventos causadores de impasses nos cronogramas preditivos. Os agentes cooperam nas tarefas de produção utilizando um controle de consenso baseado no processamento

de eventos em períodos de tempo pré-definidos. Experimentos comprovaram que algoritmos convencionais de escalonamento preditivo, combinados com MAS e filtros para reajustamento de cronogramas possibilitam flexibilidade na produção, evitando os tipos de impasses discutidos. Os objetivos específicos relacionados com este estudo foram alcançados e dois artigos científicos foram produzidos.

O terceiro estudo desenvolveu um novo algoritmo de escalonamento preditivo que foi capaz de gerar cronogramas com *makespans* por valores máximos, adequados para FMSs com limitações de *buffer* e de transporte (BTC). O algoritmo foi chamado de FJS-BTC e pôde empregar até cinco métodos de obtenção de matrizes de eventos, de forma que a matriz de menor custo para o escalonamento é selecionada para gerar o cronograma de produção. Os cronogramas também foram reajustados com os filtros de serialização, podagem e sobreposição, para estarem livres de impasses por BTCs. Dois tipos de consenso multiagente, adaptados para operar com cronogramas FJS-BTC reajustados, possibilitaram a inserção de ordens de urgência em *runtime*. Um algoritmo baseado em janelas de tempo, chamado OFTW, foi utilizado para validar os momentos de inserção das ordens de urgência no sistema, de forma que não ocorressem *deadlocks* por BTCs e sobreposições. O sistema multiagente foi adaptado para permitir que os agentes inserissem as ordens de produção urgentes nos períodos de convergência do MAS validados pelo OFTW. Dois artigos científicos foram produzidos neste estudo. Os estudos desenvolvidos contemplaram os objetivos geral e específicos definidos na tese e produziram novas contribuições científicas na área.

6.1 TRABALHOS FUTUROS

O aproveitamento do consenso multiagente pode ser explorado em outras aplicações na manufatura. Para trabalhos futuros o consenso de grupo pode ser adaptado para convocar agentes disponíveis, de fora do grupo, para assumir tarefas de outros agentes que falharam. Por exemplo, devido à falhas ocorridas por problemas em recursos associados a um determinado agente. Isso é importante para a flexibilidade de máquina, quando o FMS dispõe de mais de uma máquina para executar as operações da máquina que falhou. Um caminho para permitir a flexibilidade de máquina é associar os agentes a grupos de máquinas, ao invés de máquinas individuais, de forma que o grupo seja composto por máquinas semelhantes, que possam executar os mesmos tipos de operações. Para adaptar esse tipo de flexibilidade é necessário que o controle selecione automaticamente, em caso de identificação de falhas, outra máquina disponível do

grupo e roteie o *job* até ela. A capacidade de *buffer* também poderá ser dimensionada de acordo com a capacidade de *buffer* disponível nas máquinas do grupo.

Um controle de consenso com novos *buffers* no ambiente de produção também precisa ser avaliado. Um *buffer* pode ser visto como um espaço destinado ao armazenamento temporário de peças ou produtos, para absorver flutuações de tempos de processos. Logo, o aproveitamento do espaço disponível para aumento do tamanho do *buffer* no ambiente de produção pode minimizar bloqueios ou erros, evitando possíveis quedas de produtividade e otimizar o *makespan* (agilizando o fornecimento). Além disso, os *buffers* também poderão ser utilizado paras absorver pequenas paradas de máquinas para não bloquear a produção que a antecede.

Uma outra possibilidade de aproveitamento do consenso está relacionada com o sincronismo, que ocorre somente quando a operação mais demorada do grupo de agentes com operações paralelas é finalizada. A espera pelo término da operação mais demorada ocasiona períodos de indisponibilidade das máquinas que finalizaram operações mais curtas. Apesar de ser uma característica prevista, devido à espera pela disponibilidade do sistema de transporte em recursos de *buffer* nulo, o incremento de sincronismo poderia ser forçado via controle nos casos em que as operações independem do transporte (por exemplo, em eventos com recursos de *buffer* limitado). Ou ainda, considerando que o sistema de transporte esteja disponível, ou que haja redundância no transporte, os períodos de ociosidade poderiam ser aproveitados para a inclusão de tarefas de produção mais curtas, dentro da janela de tempo da indisponibilidade dos recursos de produção com operações mais demoradas, a fim de otimizar o uso dos demais recursos disponíveis no ambiente de produção.

Em estudos futuros, novos filtros de reajustamento podem ser desenvolvidos para lidar com a relação entre a trajetória e o tempo de processamento dos produtos, objetivando reduzir o *makespan*. Entretanto, alguns problemas são difíceis de resolver no reajustamento dos cronogramas de produção (p. ex., imprevistos em tempo de execução). Nestes casos, o reescalonamento pode ser o foco da pesquisa e Dempster-Shafer, juntamente com tecnologias da Indústria 4.0, podem melhorar as tomadas de decisões em reescalonamentos. O principal desafio nos processos de decisão é a disponibilidade de informações que apoiem a compreensão dos problemas e a escolha de soluções. O controle de consenso proposto possibilita que os agentes tenham conhecimento sobre o estado do sistema a cada evolução de eventos, característica que poderá ser aproveitada para garantir as informações necessárias ao reescalonamento.

Uma outra proposta de trabalho futuro é capacitar os agentes a obterem consenso parcial

do grupo, em situações onde o reescalonamento é necessário. Por exemplo, no caso de um agente não reportar o término de uma tarefa e ela já tenha expirado (encerrado o seu tempo de conclusão), indicando uma possível falha de máquina ou comunicação. Logo, o sistema deverá possibilitar que outras ações de controle sejam tomadas para a continuidade da produção. Entretanto, à medida que um sistema cresce em complexidade, os agentes podem ter dificuldades de se organizar, gerenciar e melhorar a eficiência do sistema. Ferramentas para sistemas distribuídos podem auxiliar na tolerância à falhas e em problemas de comunicação.

REFERÊNCIAS

ABBASI, Milad Haji; MAJIDI, Babak; MANZURI, Mohammad Taghi. Glimpse-gaze deep vision for modular rapidly deployable decision support agent in smart jungle. *In*: **2018 6th Iranian Joint Congress on Fuzzy and Intelligent Systems (CFIS)**. *[S.l.: s.n.]*, 2018. p. 75–78. ISSN null.

ABDOLRAZZAGH-NEZHAD, Majid; ABDULLAH, Salwani. Job shop scheduling: Classification, constraints and objective functions. **International Journal of Computer, Electrical, Automation, Control and Information Engineering**, n. 4, 09 2017.

ABIDIN, Z.Z.; HAMZAH, M.S.M.; ARSHAD, Mohd Rizal; NGAH, Umi. A calibration framework for swarming asvs' system design. **Indian Journal of Marine Sciences**, v. 41, p. 581–588, 02 2012.

AFANASYEV, Ilya; KOLOTOV, Alexander; REZIN, Ruslan; DANILOV, Konstantin; KASHEVNIK, Alexey; JOTSOV, Vladimir. **Blockchain Solutions for Multi-Agent Robotic Systems**. 2019.

AHMED, S; HASAN, M; SUBHAN, F. Optimal filtering in multi-agent robots using leader-follower formation. *In*: **17th IEEE Intl. Multi Topic Conf. 2014**. *[S.l.: s.n.]*, 2014. p. 440–444. ISSN null.

AHSAN, Muhammad; MA, Qian. Bipartite containment control of multi-agent systems. *In*: **2019 IEEE/ASME Intl. Conf. on Advanced Intelligent Mechatronics (AIM)**. *[S.l.: s.n.]*, 2019. p. 895–900. ISSN 2159-6247.

AMIRKHANI, Abdollah; BARSHOOI, Amir Hossein. Consensus in multi-agent systems: a review. **Artificial Intelligence Review**, Springer, v. 55, n. 5, p. 3897–3935, 2022.

BAER, Schirin; BAKAKEU, Jupiter; MEYES, Richard; MEISEN, Tobias. Multi-agent reinforcement learning for job shop scheduling in flexible manufacturing systems. *In*: **2019 Second International Conference on Artificial Intelligence for Industries (AI4I)**. *[S.l.: s.n.]*, 2019. p. 22–25.

BARENJI, Reza Vatankhah; BARENJI, Ali Vatankhah; HASHEMIPOUR, Majid. A multi-agent rfid-enabled distributed control system for a flexible manufacturing shop. **The International Journal of Advanced Manufacturing Technology**, v. 71, n. 9, p. 1773–1791, Apr 2014. ISSN 1433-3015. Disponível em: https://doi.org/10.1007/s00170-013-5597-2.

BASHIR, Muhammad. Optimal enforcement of liveness for decentralized systems of flexible manufacturing systems using petri nets. **Transactions of the Institute of Measurement and Control**, v. 42, n. 12, p. 2206–2220, 2020.

BI, Mingjie; KOVALENKO, Ilya; TILBURY, Dawn M.; BARTON, Kira. Dynamic resource allocation using multi-agent control for manufacturing systems∗∗this work was funded in part by nsf 1544678. **IFAC-PapersOnLine**, v. 54, n. 20, p. 488–494, 2021. ISSN 2405-8963. Modeling, Estimation and Control Conference MECC 2021.

BOCCELLA, Anna Rosaria; CENTOBELLI, Piera; CERCHIONE, Roberto; MURINO, Teresa; RIEDEL, Ralph. Evaluating centralized and heterarchical control of smart manufacturing systems in the era of industry 4.0. **Applied Sciences**, v. 10, n. 3, 2020. ISSN 2076-3417.

BORISOV, Oleg I.; GROMOV, Vladislav S.; KOLYUBIN, Sergey A.; PYRKIN, Anton A. Case study on human-free water heaters production for i4.0. *In*: **2018 IEEE Industrial Cyber-Physical Systems (ICPS)**. *[S.l.: s.n.]*, 2018. p. 369–374.

BOUTERAA, Yassine; DERBEL, Nabil. Distributed second order sliding mode control for networked robots synchronisation: theory and experimental results. **Intl. Journal of Modelling, Ident. and Control**, v. 29, p. 90, 01 2018.

BRAHMI, Abdelkrim; SAAD, Maarouf; GAUTHIER, Guy; ZHU, Wen-Hong; GHOMMAM, Jawhar. Adaptive backstepping control of multi-mobile manipulators handling a rigid object in coordination. **International Journal of Modelling, Identification and Control**, v. 31, n. 2, p. 169–181, 2019.

CALDEIRA, Rylan H.; GNANAVELBABU, A.; VAIDYANATHAN, T. An effective backtracking search algorithm for multi-objective flexible job shop scheduling considering new job arrivals and energy consumption. **Computers & Industrial Engineering**, v. 149, p. 106863, 2020. ISSN 0360-8352.

CALà, Ambra; RYASHENTSEVA, Daria; LüDER, Arndt. Modeling approach for a flexible manufacturing control system. *In*: **2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)**. *[S.l.: s.n.]*, 2016. p. 1–4.

CHAUDHRY, Imran Ali; KHAN, Abid Ali. A research survey: review of flexible job shop scheduling techniques. **International Transactions in Operational Research**, v. 23, n. 3, p. 551–591, 2016.

CHEUNG, Yushing; CHUNG, Jae; PATEL, Ketula. Semi-autonomous collaborative control of multi-robotic systems for multi-task multi-target pairing. *In*: . *[S.l.: s.n.]*, 2011. v. 7.

CHOUHAN, Satyendra Singh; NIYOGI, Rajdeep. An analysis of the effect of communication for multi-agent planning in a grid world domain. **International Journal of Intelligent Systems and Applications**, Citeseer, v. 4, n. 5, p. 8, 2012.

DABAH, Adel; BENDJOUDI, Ahcene; AITZAI, Abdelhakim; TABOUDJEMAT, Nadia Nouali. Efficient parallel tabu search for the blocking job shop scheduling problem. **Soft Computing**, v. 23, p. 13283–13295, 2019.

DARINTSEV, O.V.; YUDINTSEV, B.S.; ALEKSEEV, A.Yu.; BOGDANOV, D.R.; MIGRANOV, A.B. Methods of a heterogeneous multi-agent robotic system group control. **Procedia Computer Science**, v. 150, p. 687 – 694, 2019. ISSN 1877-0509. Proceedings of the 13th Intl. Symposium "Intelligent Systems 2018" (INTELS'18), 22-24 October, 2018, St. Petersburg, Russia.

DENG, Chao; YANG, Guang-Hong. Leaderless and leader-following consensus of linear multi-agent systems with distributed event-triggered estimators. **Journal of the Franklin Institute**, v. 356, n. 1, p. 309–333, 2019. ISSN 0016-0032.

DENKENA, Berend; DITTRICH, Marc-André; FOHLMEISTER, Silas; KEMP, Daniel; PALMER, Gregory. Scalable cooperative mulit-agent-reinforcment-learning for order-controlled on schedule manufacturing in flexible manufacturing systems. *In*: FRANKE, Jörg; SCHUDERER, Peter (Ed.). **Simulation in Produktion und Logistik 2021**. Göttingen: Cuvillier Verlag, 2021. p. 305–314. ISBN 9783736974791.

DIAZ, Jenny L.; OCAMPO-MARTINEZ, Carlos. Non-centralised control strategies for energy-efficient and flexible manufacturing systems. **Journal of Manufacturing Systems**, v. 59, p. 386–397, 2021. ISSN 0278-6125.

DORRI, Ali; KANHERE, Salil S; JURDAK, Raja. Multi-agent systems: A survey. **IEEE Access**, v. 6, p. 28573–28593, 2018.

DOUSH, Iyad Abu; AL-BETAR, Mohammed Azmi; AWADALLAH, Mohammed A.; SANTOS, Eugene; HAMMOURI, Abdelaziz I.; MAFARJEH, Majdi; ALMERAJ, Zainab. Flow shop scheduling with blocking using modified harmony search algorithm with neighboring heuristics methods. **Applied Soft Computing**, v. 85, p. 105861, 2019. ISSN 1568-4946.

DU, N.; HU, H.; ZHOU, M. Robust deadlock avoidance and control of automated manufacturing systems with assembly operations using petri nets. **IEEE Transactions on Automation Science and Engineering**, v. 17, n. 4, p. 1961–1975, 2020.

DUCHOň, František; VONDRáčEK, Martin; DEKAN, Martin; BABINEC, Andrej; SPIELMANN, Róbert; SZABOVá, Martina; MIKULOVá, Zuzana; BEňO, Peter; DúBRAVSKý, Jozef. Homogenous multi-robot system for mapping of unknown environment. *In*: **2016 IEEE 14th International Symposium on Applied Machine Intelligence and Informatics (SAMI)**. *[S.l.: s.n.]*, 2016. p. 17–22.

DURASEVIć, Marko; JAKOBOVIć, Domagoj. Comparison of schedule generation schemes for designing dispatching rules with genetic programming in the unrelated machines environment. **Applied Soft Computing**, v. 96, p. 106637, 2020. ISSN 1568-4946.

EFREMOV, Mikhail A; KHOLOD, Ivan I. Swarm robotics foraging approaches. *In*: IEEE. **2020 IEEE Conf. of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)**. *[S.l.]*, 2020. p. 299–304.

FAR, Mohammad Hemmati; HALEH, Hassan; SAGHAEI, Abbas. A flexible cell scheduling problem with automated guided vehicles and robots under energy-conscious policy. **Scientia Iranica**, Sharif University of Technology, v. 25, n. 1, p. 339–358, 2018. ISSN 1026-3098.

FAZLIRAD, A.; BRENNAN, R. W. Multiagent manufacturing scheduling: An updated state of the art review. *In*: **2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)**. *[S.l.: s.n.]*, 2018. p. 722–729.

FIGAT, Maksym; ZIELIŃSKI, Cezary. Robotic system specification methodology based on hierarchical petri nets. **IEEE Access**, v. 8, p. 71617–71627, 2020.

FIPA. **FIPA Agent Management Specification**. Geneva, Switzerland, 2004. Supersedes FIPA00002, FIPA00017, FIPA00019.

FORGHANI, Kamran; Fatemi Ghomi, S.M.T. Joint cell formation, cell scheduling, and group layout problem in virtual and classical cellular manufacturing systems. **Applied Soft Computing**, p. 106719, 2020. ISSN 1568-4946.

FRANCESCHELLI, Mauro; GASPARRI, Andrea; GIUA, Alessandro; ULIVI, Giovanni. Decentralized stabilization of heterogeneous linear multi-agent systems. *In*: **2010 IEEE International Conference on Robotics and Automation**. *[S.l.: s.n.]*, 2010. p. 3556–3561.

GANZHUR, A. P.; GANZHUR, M. A.; KOBYLKO, A. E.; DEMCHENKOVA, M. N. Development of information security in the ”internet of things”. **AIP Conference Proceedings**, v. 2188, n. 1, p. 050022, 2019.

GAO, Zhenxin; FENG, Yanxiang; XING, Keyi. A hybrid estimation-of-distribution algorithm for scheduling flexible job shop with limited buffers based on petri nets. **IEEE Access**, v. 8, p. 165396–165408, 2020.

GARCíA, Cecilia; CáRDENAS, Pedro F.; PUGLISI, Lisdandro J.; SALTAREN, Roque. Design and modeling of the multi-agent robotic system: Smart. **Robotics and Autonomous Systems**, v. 60, n. 2, p. 143–153, 2012. ISSN 0921-8890.

GEHLHOFF, Felix; FAY, Alexander. On agent-based decentralized and integrated scheduling for small-scale manufacturing. **at - Automatisierungstechnik**, v. 68, n. 1, p. 15–31, 2020.

GIL, Angel; AGUILAR, Jose; DAPENA, Eladio; RIVAS, Rafael. A control architecture for robot swarms (ameb). **Cybernetics and Systems**, Taylor & Francis, v. 50, n. 3, p. 300–322, 2019.

GROOVER, Mikell P. **Fundamentals of modern manufacturing: materials, processes, and systems**. *[S.l.]*: John Wiley & Sons, 2020.

HAJDUK, M.; SUKOP, M.; SEMJON, J. Principles of formation of flexible manufacturing system. **Technical gazette**, v. 25, n. 3, p. 649–654, 2018.

HAM, Andy. Transfer-robot task scheduling in job shop. **Intl. Journal of Production Research**, Taylor & Francis, v. 0, n. 0, p. 1–11, 2020.

HAN, Xiao; WANG, Zili; HE, Yihai; ZHAO, Yixiao; CHEN, Zhaoxiang; ZHOU, Di. A mission reliability-driven manufacturing system health state evaluation method based on fusion of operational data. **Sensors**, v. 19, n. 3, 2019. ISSN 1424-8220.

HAN, Zhonghua; LIU, Yuehan; SHI, Haibo; TIAN, Xutian. A dynamic buffer reservation method based on markov chain to solve deadlock problem in scheduling. *In*: WANG, Rui; CHEN, Zengqiang; ZHANG, Weicun; ZHU, Quanmin (Ed.). **Proceedings of the 11th International Conference on Modelling, Identification and Control (ICMIC2019)**. Singapore: Springer Singapore, 2020. p. 1205–1213. ISBN 978-981-15-0474-7.

HANSMANN, Karl-Werner; HOECK, Michael. Production control of a flexible manufacturing system in a job shop environment. **International transactions in operational research**, Elsevier, v. 4, n. 5-6, p. 341–351, 1997.

HE, Xingyun; SHI, Lei. Automatic aid for robot control system design. *In*: **2015 International Conference on Advanced Robotics (ICAR)**. *[S.l.: s.n.]*, 2015. p. 434–439.

HU, Liang; LIU, Zhenyu; HU, Weifei; WANG, Yueyang; TAN, Jianrong; WU, Fei. Petri-net-based dynamic scheduling of flexible manufacturing system via deep reinforcement learning with graph convolutional network. **Journal of Manufacturing Systems**, v. 55, p. 1–14, 2020. ISSN 0278-6125.

HU, M.; YANG, S.; CHEN, Y. Partial reachability graph analysis of petri nets for flexible manufacturing systems. **IEEE Access**, v. 8, p. 227925–227935, 2020.

ISMAIL, Zool Hilmi; SARIFF, Nohaidda. A survey and analysis of cooperative multi-agent robot systems: Challenges and directions. *In*: HURTADO, Efren Gorrostieta (Ed.). **Applications of Mobile Robots**. Rijeka: IntechOpen, 2019. cap. 1, p. 1–22.

JAVAID, Mohd; HALEEM, Abid; SINGH, Ravi Pratap; SUMAN, Rajiv. Enabling flexible manufacturing system (FMS) through the applications of Industry 4.0 technologies. **Internet of Things and Cyber-Physical Systems**, v. 2, p. 49–62, 2022. ISSN 2667-3452.

JI, Lianghao; TONG, Shuo; LI, Huaqing. Dynamic group consensus for delayed heterogeneous multi-agent systems in cooperative-competitive networks via pinning control. **Neurocomputing**, v. 443, p. 1–11, 2021. ISSN 0925-2312.

JIMéNEZ, Andrés; DíAZ, Vicente García; BOLAñOS, Sandro. A decentralized framework for multi-agent robotic systems. **Sensors**, v. 18, p. 417, 02 2018.

KAID, Husam; AL-AHMARI, Abdulrahman; LI, Zhiwu; DAVIDRAJUH, Reggie. Single controller-based colored petri nets for deadlock control in automated manufacturing systems. **Processes**, v. 8, n. 1, 2020. ISSN 2227-9717.

KALEMPA, Vivian Cremer; PIARDI, Luis; LIMEIRA, Marcelo; OLIVEIRA, André Schneider De. Fault-resilient collective ternary-hierarchical behavior to smart factories. **IEEE Access**, v. 8, p. 176905–176915, 2020.

KALEMPA, Vivian Cremer; PIARDI, Luis; LIMEIRA, Marcelo; OLIVEIRA, André Schneider de. Multi-robot preemptive task scheduling with fault recovery: A novel approach to automatic logistics of smart factories. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 21, n. 19, p. 6536, 2021.

KAPITONOV, Aleksandr; BERMAN, Ivan; BULATOV, Vitaly; LONSHAKOV, Sergey; KRUPENKIN, Aleksandr. Protocol for organization of a decentralized autonomous agents network in factories using market mechanisms. **Intl. J. of Mechanical Engineering and Robotics Research**, Int. J. Mech. Eng. Rob. Res, v. 8, n. 5, 9 2019.

KARIGIANNIS, John; REKATSINAS, Theodoros; TZAFESTAS, Costas. Developmental learning of cooperative robot skills: A hierarchical multi-agent architecture. *In*: ____. *[S.l.: s.n.]*, 2011. p. 497–538.

KHAMLICHI, H.; OUFASKA, K.; ZOUADI, T.; DKIOUAK, R. A hybrid grasp algorithm for an integrated production planning and a group layout design in a dynamic cellular manufacturing system. **IEEE Access**, v. 8, p. 162809–162818, 2020.

KIANPOUR, Parsa; GUPTA, Deepak; KRISHNAN, Krishna Kumar; GOPALAKRISHNAN, Bhaskaran. Automated job shop scheduling with dynamic processing times and due dates using

project management and industry 4.0. **Journal of Industrial and Production Engineering**, Taylor & Francis, v. 0, n. 0, p. 1–14, 2021.

LANGE, Julia; WERNER, Frank. On neighborhood structures and repair techniques for blocking job shop scheduling problems. **Algorithms**, v. 12, n. 11, 2019. ISSN 1999-4893.

LAVENDELIS, Egons; LIEKNA, Aleksis; NIKITENKO, Agris; GRABOVSKIS, Arvids; GRUNDSPENKIS, Janis. Multi-agent robotic system architecture for effective task allocation and management. *In*: **11th WSEAS Intl. Conf. on Signal Processing, Robotics and Automation, At Cambridge, UK**. *[S.l.: s.n.]*, 2012.

LEE, Dong-Kyu; SHIN, Jeong-Hoon; LEE, Dong-Ho. Operations scheduling for an advanced flexible manufacturing system with multi-fixturing pallets. **Computers & Industrial Engineering**, v. 144, p. 106496, 2020. ISSN 0360-8352.

LI, Jun qing; DENG, Jia wen; LI, Cheng you; HAN, Yu yan; TIAN, Jie; ZHANG, Biao; WANG, Cun gang. An improved jaya algorithm for solving the flexible job shop scheduling problem with transportation and setup times. **Knowledge-Based Systems**, v. 200, p. 106032, 2020. ISSN 0950-7051.

LI, Kexin; DENG, Qianwang; ZHANG, Like; FAN, Qing; GONG, Guiliang; DING, Sun. An effective mcts-based algorithm for minimizing makespan in dynamic flexible job shop scheduling problem. **Computers & Industrial Engineering**, v. 155, p. 107211, 2021. ISSN 0360-8352.

LI, Xixing; GUO, Xing; TANG, Hongtao; WU, Rui; WANG, Lei; PANG, Shibao; LIU, Zhengchao; XU, Wenxiang; LI, Xin. Survey of integrated flexible job shop scheduling problems. **Computers & Industrial Engineering**, Elsevier, p. 108786, 2022.

LI, Yanjiang; TAN, Chong. A survey of the consensus for multi-agent systems. **Systems Science & Control Engineering**, Taylor & Francis, v. 7, n. 1, p. 468–482, 2019.

LIANG, Xu; HUANG, Yiming; HUANG, Ming. Prediction of optimal rescheduling mode of flexible job shop under the arrival of a new job. *In*: **2020 IEEE 8th International Conference on Computer Science and Network Technology (ICCSNT)**. *[S.l.: s.n.]*, 2020. p. 55–58.

LIEKNA, Aleksis; LAVENDELIS, Egons; NIKITENKO, Agris. Challenges in development of real time multi-robot system using behaviour based agents. *In*: **Distributed Computing and AI**. *[S.l.]*: Springer Intl. Pub., 2013. p. 587–595. ISBN 978-3-319-00551-5.

LIU, Min; YAO, Xifan; LI, Yongxiang. Hybrid whale optimization algorithm enhanced with lévy flight and differential evolution for job shop scheduling problems. **Applied Soft Computing**, Elsevier, v. 87, p. 105954, 2020.

LUO, J.; ZHOU, M.; WANG, J. Q. Ab&b: An anytime branch and bound algorithm for scheduling of deadlock-prone flexible manufacturing systems. **IEEE Transactions on Automation Science and Engineering**, p. 1–11, 2020.

LUO, Shu. Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning. **Applied Soft Computing**, v. 91, p. 106208, 2020. ISSN 1568-4946.

LUO, Yu-Juan; LIU, Cheng-Lin. Consensus tracking by iterative learning control for linear heterogeneous multiagent systems based on fractional-power error signals. **Algorithms**, MDPI AG, v. 12, n. 9, p. 185, Sep 2019. ISSN 1999-4893.

MALASCHUK, Olesia B.; DYMIN, Alexander A. Ad-hoc protocol for drones coordination in urban environment. *In*: **2020 Moscow Workshop on Electronic and Networking Technologies (MWENT)**. *[S.l.: s.n.]*, 2020. p. 1–3.

MAREDDY, Padma Lalitha; NARAPUREDDY, Sivarami Reddy; DWIVEDULA, Venkata Ramamurthy; KARANAM, Prahlada Rao. Development of scheduling methodology in a multi-machine flexible manufacturing system without tool delay employing flower pollination algorithm. **Engineering Applications of Artificial Intelligence**, v. 115, p. 105275, 2022. ISSN 0952-1976.

MAVROGIANNIS, Christoforos; KNEPPER, Ross A. Hamiltonian coordination primitives for decentralized multiagent navigation. **The International Journal of Robotics Research**, v. 40, n. 10-11, p. 1234–1254, 2021.

MEILANITASARI, Prita; SHIN, Seung-Jun. A review of prediction and optimization for sequence-driven scheduling in job shop flexible manufacturing systems. **Processes**, v. 9, n. 8, 2021. ISSN 2227-9717.

MESSINIS, Sotirios; VOSNIAKOS, George Christopher. An agent-based flexible manufacturing system controller with petri-net enabled algebraic deadlock avoidance. **Reports in Mechanical Engineering**, v. 1, n. 1, p. 77–92, Oct. 2020.

MEZGEBE, Tsegay Tesfay; HAOUZI, Hind Bril El; DEMESURE, Guillaume; PANNEQUIN, Remi; THOMAS, Andre. Multi-agent systems negotiation to deal with dynamic scheduling in disturbed industrial context. **Journal of Intelligent Manufacturing**, Springer, v. 31, n. 6, p. 1367–1382, 2020.

MIAH, Md Suruz; NGUYEN, Bao; BOURQUE, Alex; SPINELLO, Davide. Non-autonomous area coverage and coordination of a multi-agent system for harbor protection applications. *In*: . *[S.l.: s.n.]*, 2018. p. 486–492.

MIYAWAKI, Masaya; MORIYAMA, Koichi; MUTOH, Atsuko; MATSUI, Tohgoroh; INUZUKA, Nobuhiro. Evolution direction of reward appraisal in reinforcement learning agents. *In*: JEZIC, Gordan; CHEN-BURGER, Yun-Heh Jessica; HOWLETT, Robert J.; JAIN, Lakhmi C.; VLACIC, Ljubo; ŠPERKA, Roman (Ed.). **Agents and Multi-Agent Systems: Technologies and Applications 2018**. Cham: Springer International Publishing, 2018. p. 13–22. ISBN 978-3-319-92031-3.

MOGALI, Jayanth Krishna; BARBULESCU, Laura; SMITH, Stephen F. Efficient primal heuristic updates for the blocking job shop problem. **European Journal of Operational Research**, v. 295, n. 1, p. 82–101, 2021. ISSN 0377-2217.

MOURTZIS, Dimitris; ANGELOPOULOS, John; DIMITRAKOPOULOS, George. Design and development of a flexible manufacturing cell in the concept of learning factory paradigm for the education of generation 4.0 engineers. **Procedia Manufacturing**, v. 45, p. 361–366, 2020. ISSN 2351-9789. Learning Factories across the value chain – from innovation to service – The 10th Conference on Learning Factories 2020.

NABI, H.Z.; AIZED, T. Performance evaluation of a carousel configured multiple products flexible manufacturing system using petri net. **Operations Management Research**, v. 13, p. 109–129, 2020.

NAJJAR, Lubna; JOHARI, Noor; QAMHIEH, Manar. A leader-follower communication protocol for multi-agent robotic systems. *In*: **2019 IEEE Jordan Intl. Joint Conf. on Electrical Engineering and Information Technology (JEEIT)**. *[S.l.: s.n.]*, 2019. p. 742–747.

NORRAZI, M. Azrai M.; YAP, Wee Yang Ricky; SANHOURY. Formations strategies for obstacle avoidance with multi agent robotic system. *In*: **Intelligent Robotics Systems: Inspiring the NEXT**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 232–245. ISBN 978-3-642-40409-2.

OH, Kwang-Kyo; PARK, Myoung-Chul; AHN, Hyo-Sung. A survey of multi-agent formation control. **Automatica**, v. 53, p. 424 – 440, 2015. ISSN 0005-1098.

OMAR, AL-Buraiki; PAYEUR, Pierre. Probabilistic task assignment for specialized multi-agent robotic systems. *In*: **2019 IEEE Intl. Symposium on Robotic and Sensors Environments (ROSE)**. *[S.l.: s.n.]*, 2019. p. 1–7.

PABON, Juan D; MOJICA-NAVA, Eduardo. Event-triggered coordination of multi-agent systems in agricultural environments. *In*: **2019 IEEE 4th Colombian Conf. on Automatic Control (CCAC)**. *[S.l.: s.n.]*, 2019. p. 1–6. ISSN null.

PARK, Myeongjin; KIM, Kihoon; KWON, Ohmin. Quantized consensus criterion for discrete-time multi-agent systems with communication delay. *In*: **2013 13th International**

Conference on Control, Automation and Systems (ICCAS 2013). *[S.l.: s.n.]*, 2013. p. 1656–1660.

PEREVERZEVA, Inna; TROUBITSYNA, Elena; LAIBINIS, Linas. Development of fault tolerant mas with cooperative error recovery by refinement in event-b. **arXiv e-prints**, p. arXiv:1210.7035, oct 2012.

PERRON, Laurent; FURNON, Vincent. **OR-Tools**. 2021. V9.1 (version 9.1), Google. Disponível em: https://developers.google.com/optimization/.

PIARDI, Luis; KALEMPA, Vivian Cremer; LIMEIRA, Marcelo; OLIVEIRA, André Schneider de; LEITÃO, Paulo. Arena—augmented reality to enhanced experimentation in smart warehouses. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 19, n. 19, p. 4308, 2019.

PIERPAOLI, Pietro; SAUTER, Dominique; EGERSTEDT, Magnus. Fault tolerant control for networked mobile robots. *In*: **2018 IEEE Conference on Control Technology and Applications (CCTA)**. *[S.l.: s.n.]*, 2018. p. 374–379.

PINEDO, L. Michael. **Scheduling. Theory, Algorithms and systems**. *[S.l.]*: Springer, 2022. ISBN 978-3-031-05920-9.

PULJIZ, David; VARGA, Maja; BOGDAN, Stjepan. Stochastic search strategies in 2d using agents with limited perception. **IFAC Proceedings Volumes**, v. 45, n. 22, p. 650 – 654, 2012. ISSN 1474-6670. 10th IFAC Symposium on Robot Control.

QUINTERO, M.; CHRISTIAN, G.; LóPEZ, José Oñate; BERTEL, R. Intelligent exploration and surveillance algorithms for multi-agents robotics systems. *In*: **2012 XXXVIII Conferencia Latinoamericana En Informatica (CLEI)**. *[S.l.: s.n.]*, 2012. p. 1–10.

RAHIMI, Reihane; ABDOLLAHI, Farzaneh; NAQSHI, Karo. Time-varying formation control of a collaborative heterogeneous multi agent system. **Robotics and Autonomous Systems**, v. 62, n. 12, p. 1799 – 1805, 2014. ISSN 0921-8890.

REINEKING, Thomas. **Belief Functions: Theory and Algorithms**. 2014. Tese (Doctoral dissertation) — Universität Bremen, 2014.

REZAEE, Hamed; ABDOLLAHI, Farzaneh. A cyclic pursuit framework for networked mobile agents based on vector field approach. **Journal of the Franklin Institute**, v. 356, n. 2, p. 1113 – 1130, 2019. ISSN 0016-0032.

ROSA, Fernando De la; ROMERO, Germán; BONILLA, Freddys. Control architecture for cooperative mobile robotic tasks. *In*: **2015 12th Latin American Robotics Symposium and**

2015 3rd Brazilian Symposium on Robotics (LARS-SBR). *[S.l.: s.n.]*, 2015. p. 79–84. ISSN null.

ROSS, Matt; PAYEUR, Pierre; CHARTIER, Sylvain. Task allocation for heterogeneous robots using a self-organizing contextual map. *In*: **2019 IEEE Intl. Symposium on Robotic and Sensors Environments (ROSE)**. *[S.l.: s.n.]*, 2019. p. 1–6.

RUSSELL, Stuart; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2016. ISBN 1292153962, 9781292153964.

SADEGHI, Azadeh; SUER, Gursel; SINAKI, Roohollah Younes; WILSON, Darin. Cellular manufacturing design and replenishment strategy in a capacitated supply chain system: A simulation-based analysis. **Computers & Industrial Engineering**, v. 141, p. 106282, 2020. ISSN 0360-8352.

SASSO, Veronica Dal; LAMORGESE, Leonardo; MANNINO, Carlo; ONOFRI, Andrea; VENTURA, Paolo. The tick formulation for deadlock detection and avoidance in railways traffic control. **Journal of Rail Transport Planning & Management**, v. 17, p. 100239, 2021. ISSN 2210-9706.

SELLITTO, Miguel Afonso. Analysis of maintenance policies supported by simulation in a flexible manufacturing cell. **Ingeniare. Revista chilena de ingeniería**, v. 28, p. 293–303, 2020. ISSN 0718-3305.

SEREBRENNY, Vladimir; LAPIN, Dmitriy; MOKAEVA, A.; SHEREUZHEV, M. Technological collaborative robotic systems. *In*: . *[S.l.: s.n.]*, 2019. v. 2171, p. 170008.

SHAFER, Glenn. **A Mathematical Theory of Evidence**. 1. ed. *[S.l.]*: Princeton University Press, 1976. ISBN 0-691-08175-1.

SHAO, Nuan; LI, Huiguang; WU, Xueli; LI, Guoyou. Distributed containment control and state observer design for multi-agent robotic system. **Intl. Journal of Modelling, Identification and Control**, v. 23, p. 193, 06 2015.

SHAW, Samuel; WENZEL, Emerson; WALKER, Alexis; SARTORETTI, Guillaume. Formic: Foraging via multiagent rl with implicit communication. **IEEE Robotics and Automation Letters**, v. 7, n. 2, p. 4877–4884, 2022.

SHUMSKAYA, Olga; ISKHAKOVA, Anastasia. Application of digital watermarks in the problem of operating signal hidden transfer in mars. *In*: **2019 Intl. Siberian Conf. on Control and Commn. (SIBCON)**. *[S.l.: s.n.]*, 2019. p. 1–5.

SHUTIN, Dmitriy; ZHANG, Siwei. Distributed sparsity-based bearing estimation with a swarm of cooperative agents. *In*: **2016 IEEE Global Conf. on Signal and Information Processing (GlobalSIP)**. *[S.l.: s.n.]*, 2016. p. 555–559. ISSN null.

SINGH, Abhinav; SHARMA, Harshita; SINGH, Neetu; KATYAL, Poonam. A survey on cooperative mobile robotics. **International Journal of Applied Engineering Research**, Research India Publications, v. 13, n. 7, p. 5160–5166, 2018. ISSN 0973-4562.

SINGHAL, Aniruddha; SINGH, Harsh Vardhan; PENUMATSA, Aarathi; BHATT, Nakul; AMBWANI, Prakash; KUMAR, Swagat; SINHA, Rajesh. An actor based architecture for multi-robot system with application to warehouse. *In*: **Proceedings of the 1st Intl. Workshop on Internet of People, Assistive Robots and Things**. New York, NY, USA: Association for Comp. Machinery, 2018. p. 13–18. ISBN 9781450358439.

SORIANO, Angel; BERNABEU, Enrique J.; VALERA, Angel; VALLÉS, Marina. Collision avoidance of mobile robots using multi-agent systems. *In*: OMATU, Sigeru; NEVES, José; RODRIGUEZ, Juan M. Corchado; SANTANA, Juan F Paz; GONZALEZ, Sara Rodríguez (Ed.). **Distributed Computing and Artificial Intelligence**. Cham: Springer Intl. Publishing, 2013. p. 429–437. ISBN 978-3-319-00551-5.

SOUMYA, S.; GURUPRASAD, K. R. Multi-agent system inspired distributed control of a serial-link robot. **Journal of Automation Mobile Robotics and Intelligent Systems**, Vol. 14, No. 1, p. 29–38, 2020. ISSN 1897-8649.

SOUSA, Alex L.; OLIVEIRA, André S. Distributed MAS with leaderless consensus to job-shop scheduler in a virtual smart factory with modular conveyors. *In*: **2020 Latin American Robotics Symposium (LARS), 2020 Brazilian Symposium on Robotics (SBR) and 2020 Workshop on Robotics in Education (WRE)**. *[S.l.: s.n.]*, 2020. p. 1–6.

SOUSA, Alex L.; OLIVEIRA, André S. Order-controlled production employing multi-agent and flexible job-shop scheduling on a physical simulation platform. *In*: **2022 Latin American Robotics Symposium (LARS), 2022 Brazilian Symposium on Robotics (SBR), and 2022 Workshop on Robotics in Education (WRE)**. *[S.l.: s.n.]*, 2022. p. 229–234.

SOUSA, Alex L.; OLIVEIRA, André S. Finite-time consensus and readjustment three-stage filter for predictive schedules in FMS. **IEEE Access**, 2023.

SRIVASTAVA, Siddharth; SARKAR, Aritra; MANOJ, BS. Hazard control algorithms for heterogenous multi-agent cloud-enabled robotic network. *In*: **2013 IEEE Intl. Conf. on Advanced Networks and Telecommunications Systems (ANTS)**. *[S.l.: s.n.]*, 2013. p. 1–6. ISSN 2153-1684.

STOLLEIS, Karl. **The Ant and the Trap: Evolution of Ant-Inspired Obstacle Avoidance in a Multi-Agent Robotic System**. 5 2015. Dissertação (Mestrado) — University of New Mexico, Albuquerque, New Mexico, 5 2015.

SUN, Yige; CHUNG, Sai-Ho; WEN, Xin; MA, Hoi-Lam. Novel robotic job-shop scheduling models with deadlock and robot movement considerations. **Transportation Research Part E: Logistics and Transportation Review**, v. 149, p. 102273, 2021. ISSN 1366-5545.

TEYMOURIFAR, Aydin; OZTURK, Gurkan; OZTURK, Zehra Kamisli; BAHADIR, Ozan. Extracting new dispatching rules for multi-objective dynamic flexible job shop scheduling with limited buffer spaces. **Cognitive Computation**, v. 12, p. 195–205, 2020.

TING, T.; WAN, Kaiyu; MAN, Ka Lok; LEE, Sanghyuk. Space exploration of multi-agent robotics via genetic algorithm. *In*: PARK, James J.; ZOMAYA, Albert; YEO, Sang-Soo; SAHNI, Sartaj (Ed.). **Network and Parallel Comp.** Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 500–507. ISBN 978-3-642-35606-3.

UHLMANN, Iracyanne Retto; FRAZZON, Enzo Morosini. Production rescheduling review: Opportunities for industrial integration and practical applications. **Journal of Manufacturing Systems**, v. 49, p. 186–193, 2018. ISSN 0278-6125.

VIKSNIN, Ilya; CHUPROV, Sergey; USOVA, Maria; ZAKOLDAEV, Danil. Police office model for multi-agent robotic systems. **IOP Conf. Series: Materials Science and Engineering**, IOP Publishing, v. 497, p. 012036, apr 2019.

VOROTNIKOV, Sergey; ERMISHIN, Konstantin; NAZAROVA, Anaid; YUSCHENKO, Arkady. Multi-agent robotic systems in collaborative robotics. *In*: **Interactive Collaborative Robotics**. *[S.l.]*: Springer Intl. Publishing, 2018. p. 270–279. ISBN 978-3-319-99582-3.

WANG, Hongcheng; JIANG, Yuchen; WANG, Hao; LUO, Hao. An online optimization scheme of the dynamic flexible job shop scheduling problem for intelligent manufacturing. *In*: **2022 4th International Conference on Industrial Artificial Intelligence (IAI)**. *[S.l.: s.n.]*, 2022. p. 1–6.

WANG, Huaizhu; WANG, Chen. Finite-time containment control of multi-agent systems with static or dynamic leaders. **Neurocomputing**, v. 226, p. 1 – 6, 2017. ISSN 0925-2312.

WEISS, Gerhard. **Multiagent Systems**. *[S.l.]*: The MIT Press, 2013. ISBN 0262018896, 9780262018890.

WEST, Jonathan M. **Applying Heterogeneous Teams of Robotic Agents Using Hybrid Communications to Mapping and Education**. 12 2017. Tese (Doutorado) — Electrical and Computer Engineering, The University of New Mexico, 12 2017. Dissertation for the degree of Doctor of Philosophy Engineering.

WOOLDRIDGE, Michael. **An Introduction to MultiAgent Systems**. 2nd. ed. *[S.l.]*: Wiley Publishing, 2009. ISBN 0470519460, 9780470519462.

WU, Lang; ZHAO, Yaping; FENG, Yuanyue; NIU, Ben; XU, Xiaoyun. Minimizing makespan of stochastic customer orders in cellular manufacturing systems with parallel machines. **Computers & Operations Research**, v. 125, p. 105101, 2021. ISSN 0305-0548.

XIE, Jin; GAO, Liang; PENG, Kunkun; LI, Xinyu; LI, Haoran. Review on flexible job shop scheduling. **IET Collaborative Intelligent Manufacturing**, Institution of Engineering and Technology, v. 1, p. 67–77(10), September 2019.

XU, Gongdan; CHEN, Yufeng. Petri-net-based scheduling of flexible manufacturing systems using an estimate function. **Symmetry**, v. 14, n. 5, 2022. ISSN 2073-8994.

YANG, Yongliang; MODARES, Hamidreza; VAMVOUDAKIS, Kyriakos G; YIN, Yixin; WUNSCH, Donald C. Model-free event-triggered containment control of multi-agent systems. *In*: **2018 Annual American Control Conf. (ACC)**. *[S.l.: s.n.]*, 2018. p. 877–884. ISSN 2378-5861.

YAO, Zhifeng; YE, Xiufen; DAI, Xuefeng. Bidding coordination algorithm with cfc and an emotion switch. **Robotica**, Cambridge University Press, v. 36, n. 4, p. 607–623, 2018.

ZHANG, Jian; DING, Guofu; ZOU, Yisheng; QIN, Shengfeng; FU, Jianlin. Review of job shop scheduling research and its new perspectives under industry 4.0. **Journal of Intelligent Manufacturing**, p. 1809–1830, 2019.

ZHANG, Jiancheng; ZHU, Fanglai. Observer-based output consensus of a class of heterogeneous mas with unmatched disturbances. **Commn. in Nonlinear Science and Num. Simulation**, v. 56, p. 240 – 251, 2018. ISSN 1007-5704.

ZHANG, Meng; TAO, Fei; NEE, A.Y.C. Digital twin enhanced dynamic job-shop scheduling. **Journal of Manufacturing Systems**, 2020. ISSN 0278-6125.

ZHANG, Xiaogang; LI, Yulong; RAN, Yan; ZHANG, Genbao. Stochastic models for performance analysis of multistate flexible manufacturing cells. **Journal of Manufacturing Systems**, v. 55, p. 94–108, 2020. ISSN 0278-6125.

ZUO, Zongyu; HAN, Qing-Long; NING, Boda. **Fixed-Time Cooperative Control of Multi-Agent Systems**. 1st. ed. *[S.l.]*: Springer Publishing Company, Incorporated, 2019. ISBN 303020278X.

ČAPKOVIč, František. Dealing with deadlocks in industrial multi agent systems. **Future Internet**, v. 15, n. 3, 2023. ISSN 1999-5903.

APÊNDICE A – REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS DE UM AGENTE

Na Tabela 20, o requisito RF1 está relacionado com o tópico *broadcast*, os requisitos RF02 – RF07 com o tópico *synchrony*, RF08 – RF11 com *signaling* e RF12 com o *notification*. O controle não é apresentado nos requisitos, pois é exclusivo para cada tipo de recurso do FMS.

Tabela 20 – Requisitos funcionais do agente.

RF01 - Armazenar eventos do cronograma no *buffer* de agente Armazenar em uma estrutura de dados do tipo lista (ou *array*) todos os eventos recebidos do nó escalonador/-despachente, via tópico *broadcast*, para compor o crononograma de produção.
RF02 - Atualizar o valor de *token*/*sync* Quando um novo valor de *token*/*sync* é recebido no tópico *synchrony*, os agentes devem atualizar o valor do *token*/*sync* corrente.
RF03 - Verificar se evento corrente é de responsabilidade do agente Identificar se o evento corrente se refere a um recurso/máquina associado ao agente.
RF04 - Executar e excluir eventos do *buffer* de agente Executar os eventos do cronograma de acordo com o índice de sequência definido para o *token*/*sync* corrente. Ao final, o evento concluído deverá ser excluído do *buffer* de agente.
RF05 - Invocar o controle para cada operação de produção O controle é responsável por gerenciar os recursos de produção e executar, a cada evento, as operações necessárias para a fabricação das peças/produtos no FMS.
RF06 - Sinalizar a conclusão de um evento do cronograma Apenas os agentes que possuem recursos associados aos eventos relacionados com o *token*/*sync* corrente devem sinalizar a sua conclusão para posterior incremento do *token*/*sync*.
RF07 - Prover ações colaborativas para as operações de produção Quando um agente não participa diretamente de um evento, ele é chamado de coadjuvante; o coadjuvante verifica em seu controle se precisa colaborar com o agente principal no evento corrente.
RF08 - Verificar a inserção de ordens urgentes Sempre que um agente sinaliza a finalização de um evento (via tópico *signaling*), os agentes do MAS deverão checar se há uma ordem de urgência para ser executada.
RF09 - Incrementar o valor de *token*/*sync* Apenas o agente que finalizou a execução de um evento poderá incrementar o valor de *token*/*sync* para a evolução da produção (exceto os agentes coadjuvantes).
RF10 - Aguardar convergência para um estado em comum (consenso completo) O valor de *token*/*sync* somente poderá ser incrementado quando todos os agentes do MAS convergirem para um estado comum, ou seja, após a finalização de todas as operações relacionadas com o *token*/*sync* corrente.
RF11 - Aguardar convergência para um estado em comum (consenso de grupo) O valor de *token*/*sync* poderá ser incrementado quando todos os agentes responsáveis pela execução de uma ordem urgente convergirem para um estado comum, independente dos demais agentes do MAS.
RF12 - Notificação de prioridade e validação de ordens urgentes O agente deve ativar sua *flag* de prioridade quando o escalonador/despachante enviar uma notificação. A *flag* ativada implica na inserção da ordem de urgência pelo algoritmo OFTW, para posterior execução.

Tabela 21 – Requisitos não funcionais do agente.

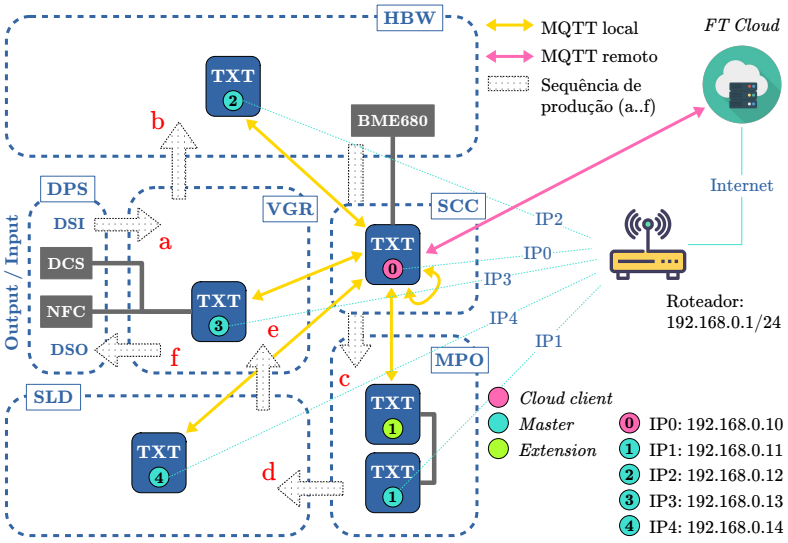
RNF01 - Infraestrutura de rede para o agente Um roteador *Wi-Fi* ou computador configurado como *hotspot* deve fornecer a rede para o MAS.
RNF02 - Suporte ao *framework* ROS Prover acesso ao ROS (instalação *desktop full*) em uma máquina virtual ou no computador onde o nó escalonador/despachante for executado (utilizar Ubuntu 20.04 e ROS Noetic).
RNF03 - Módulos de suporte para o agente Prover acesso aos seguintes módulos Python: catkin, catkin_pkg, ftrobopy, genmsg, genpy, pyparsing, roscpp, rosgraph, rosgraph_msgs, roslib, rospkg, rospy, std_msgs e yaml.

APÊNDICE B – TOPOLOGIAS DA PLATAFORMA FÍSICA DE EXPERIMENTAÇÃO

B.0.1 Topologia padrão

A topologia padrão (original) da plataforma de experimentação é centralizada, os controladores TXT se comunicam exclusivamente com um controlador TXT central (Figura 30). O controlador central (TXT-0), localizado no módulo SCC, é configurado como ”*cloud client*” e obtém endereço IP de um roteador Wi-Fi integrado na fábrica. Este controlador pode se conectar à nuvem da Fischertechnik (FT) via *bridge* MQTT remota. Na nuvem, um utilizador pode acessar via *browser* uma *dashboard* para visualizar informações e interagir em alguns processos de produção. O TXT-0 também conecta uma câmera USB (não representada na topologia) e um sensor ambiental (BME680) para dados sobre pressão do ar, umidade, temperatura e gases.

Figura 30 – Topologia padrão da plataforma de experimentação.



Fonte: Autoria própria.

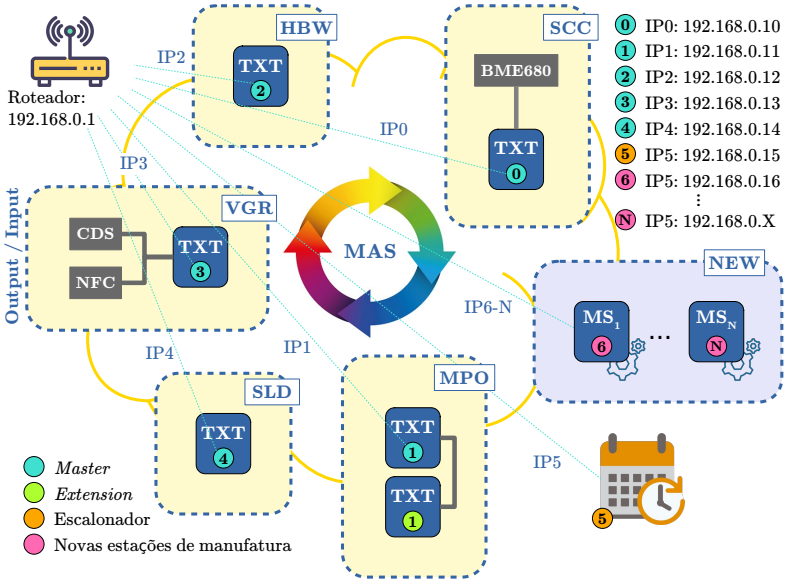
Na topologia padrão, outros quatro controladores (TXT-1 a TXT-4) são configurados como ”*master*” e também obtém IP do roteador Wi-Fi. Neste caso, a comunicação dos controladores com o TXT-0 é feita por MQTT local, ou seja, utilizando os serviços de publicação e assinatura do TXT-0 (*broker*). A estação MPO utiliza dois controladores TXT (identificados por TXT-1), um deles é configurado como ”*extension*” e controlado pelo TXT *master*. No módulo DPS estão localizados o sensor de detecção de cor (DCS) e o leitor NFC, juntamente com os locais de chegada (DSI) e retirada de peças (DSO). O módulo DPS é conectado e comandado pelo controlador TXT-3 da estação VGR, que também comanda o robô com garra de sucção a vácuo

da plataforma. Por fim, o fluxo padrão de uma peça inicia em DSI e finaliza em DSO, conforme visualizado pelas setas largas rotuladas em ordem alfabética (de ”a” até ”f”) na topologia.

B.0.2 Topologia multiagente

A nova topologia emprega sistemas multiagentes (MAS) e associa cada estação da plataforma de experimentação a um respectivo agente. Os agentes são executados nos controladores FT TXT, que se conectam em um router Wi-Fi ou computador configurado como *hotspot*. Toda a comunicação entre agentes ocorre via tópicos ROS (*Robot Operating System*) e a produção é coordenada de acordo com cronogramas definidos por um nó ROS chamado ”escalonador”. Os agentes também são nós ROS e uma visão geral da topologia é apresentada na Figura 31.

Figura 31 – Topologia multiagente da plataforma de experimentação.



Fonte: Autoria própria.

A topologia multiagente também prevê o acréscimo de outras estações de manufatura na plataforma. O bloco ”NEW” simboliza a inclusão de novas estações (de MS_1 a MS_N), como as que foram impressas em 3D para o FMS estudado (ver Capítulo 4). O MAS é capaz de lidar com diferentes sequências de produção, de acordo com cronogramas definidos por um processo de escalonamento preditivo. Os cronogramas são enviados para o MAS e cada agente processa os eventos localmente, no controlador TXT ao qual está associado.