CENTRO UNIVERSITÁRIO FEI

LEONARDO CONTADOR NEVES

SISTEMA DE PLANEJAMENTO E ACOMPANHAMENTO DE AÇÕES PARA TAREFAS DE PROPÓSITO GERAL EM ROBÓTICA DE SERVIÇO

São Bernardo do Campo

2021

LEONARDO CONTADOR NEVES

SISTEMA DE PLANEJAMENTO E ACOMPANHAMENTO DE AÇÕES PARA TAREFAS DE PROPÓSITO GERAL EM ROBÓTICA DE SERVIÇO

Dissertação de Mestrado, apresentada ao Centro Universitário da FEI para obtenção do título de Mestre em Engenharia Elétrica. Orientado pelo Prof. Dr. Plinio Thomaz Aquino Jr.

São Bernardo do Campo

2021

Contador Neves, Leonardo.
SISTEMA DE PLANEJAMENTO E ACOMPANHAMENTO DE AÇÕES PARA TAREFAS DE PROPÓSITO GERAL EM ROBÓTICA DE SERVIÇO / Leonardo Contador Neves. São Bernardo do Campo, 2021.
72 f. : il.

Dissertação - Centro Universitário FEI.
Orientador: Prof. Dr. Plinio Thomaz Aquino Jr..

1. Planejamento Probabilístico. 2. Robótica de Serviço. 3. Petri Net Plans. 4. Planeja- mento de Ações. 5. ROSPlann. I. Thomaz Aquino Jr., Plinio, orient. II. Título.

Elaborada pelo sistema de geração automática de ficha catalográfica da FEI com os dados fornecidos pelo(a) autor(a).

centro universitário FEI

APRESENTAÇÃO DE DISSERTAÇÃO
ATA DA BANCA EXAMINADORA

Programa de Pós-Graduação Stricto Sensu em Engenharia Elétrica

Mestrado

PGE-10

Aluno: Leonardo Contador Neves **Matrícula:** 118315-1

Título do Trabalho: Sistema de planejamento e acompanhamento de ações para tarefas de propósito geral em robótica de serviço.

Área de Concentração: Inteligência Artificial Aplicada à Automação e Robótica

Orientador: Prof. Dr. Plínio Thomaz Aquino Junior

Data da realização da defesa: 26/10/2021

ORIGINAL ASSINADA

Avaliação da Banca Examinadora:

A banca foi realizada no dia 26 de Outubro às 9:30 horas, com uma boa apresentação do aluno. A arguição ocorreu debatendo-se aspectos teóricos e metodológicos, onde o aluno respondeu a todas as questões de forma adequada demonstrando conhecimento pleno do tema. Cometários foram realizados para melhoria do texto em sua versão final. A aprovação ocorreu por unanimidade dos membros da banca.

São Bernardo do Campo, / / .

MEMBROS DA BANCA EXAMINADORA	
Prof. Dr. Plínio Thomaz Aquino Junior	Ass.: _____
Prof. Dr. Javier Martinez Silva	Ass.: _____
Prof. Dr. Flavio Tonidandel	Ass.: _____

A Banca Julgadora acima-assinada atribuiu ao aluno o seguinte resultado:

APROVADO ☒ REPROVADO ☐

VERSÃO FINAL DA DISSERTAÇÃO

APROVO A VERSÃO FINAL DA DISSERTAÇÃO EM QUE FORAM INCLUÍDAS AS RECOMENDAÇÕES DA BANCA EXAMINADORA

_____

Aprovação do Coordenador do Programa de Pós-graduação

_____

Prof. Dr. Carlos Eduardo Thomaz

Dedico este trabalho à meus familiares por todo o apoio e principalmente à meu avô, que mesmo não estando presente me despertou todo meu amor pela engenharia.

AGRADECIMENTOS

Em primeiro lugar, agradeço à minha família pelo apoio de sempre, agradeço ao Centro Universitário da FEI por todo o suporte quanto aos equipamentos e laboratórios que estavam sempre à disposição, agradeço à equipe ROBOFEI por anos de aprendizado e parceria, agradeço ao meu orientador pelos direcionamentos para este trabalho e anos de colaboração na equipe ROBOFEI e à todos os que estava diretamente e indiretamente me ajudando neste período, meu sincero muito obrigado. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

“A insatisfação é a principal motivadora do progresso.”

Thomas Edison

RESUMO

A cada dia um número maior de sistemas robóticos inteligentes vem fazendo parte de nosso dia à dia, sejam eles simples como robôs que limpam o chão ou porteiros eletrônicos, ou de maior complexidade como carros autônomos e robôs que auxiliam pessoas que entram em bancos. Esses sistemas inteligentes, sejam eles simples ou complexos, tem em comum uma estrutura básica independentemente das funcionalidades: Um objetivo a ser alcançado, um estado inicial e funcionalidades que o fazem alcançar seu objetivo.

Para os sistemas robóticos complexos, é fundamental que eles sejam capazes de planejar meios de atingir seus objetivos, mesmo que a princípio abstratos, e assim monitorar esse planejamento para que a estrutura seja robusta e o objetivo seja alcançado dadas as imprevisibilidades do mundo real.

Através de robôs domésticos ou de diversos outros propósitos, de tempos em tempos são vistos trabalhos aplicando técnicas de planejamento de ações para encontrar meios de criar agentes com maior inteligencia e que escolham suas ações por meio de processos deliberativos através do conhecimento prévio sobre o domínio que os rodeia. Mesmo com o esforço da comunidade que para manter essa vasta área de planejamento sempre atualizada, diversos trabalhos podem ser revisitados e melhorados, tendo em vista que novas aplicações e expansões para os algoritmos usados previamente podem ser implementadas e novos problemas podem ser introduzidos.

Neste contexto, este trabalho busca explorar o planejamento de ações para robótica doméstica, onde o *ROSPlan*, *framework* que traz módulos para planejamento de ações em robótica, é utilizado junto às Redes de Petri, para melhorar a aplicação de ambas as ferramentas em conjunto em tarefas de propósito geral em robótica de serviço. Além disso, um módulo de expansão de plano é proposto, para que possamos melhorar a eficiência da arquitetura com base no que já foi feito pela comunidade previamente. Para avaliação do modelo proposto, foram submetidos testes a partir de um simulador de tarefas da competição *Robocup@Home* e foi verificada o desempenho do robô HERA nestas execuções.

Palavras-chave: Planejamento Probabilístico, Robótica de Serviço, Petri Net Plans, Planejamento de Ações, ROSPlann

ABSTRACT

Every day more intelligent robotic systems are part of our daily lives, whether they are simple as robots that clean the floor or doorkeepers, or more complex as autonomous cars and robots that help people who enter banks. These intelligent systems, be they simpler or more complex, have in common a basic structure regardless of the functionalities: An objective to be achieved, an initial state and functionalities that make it reach its objective.

For the most complex robotic systems, it is essential that they are able to plan ways to achieve their objectives, even if at first abstract, and thus monitor that planning so that the structure is robust and the objective is achieved given the unpredictability of the real world.

Through domestic robots or various other purposes, from time to time works are seen applying action planning techniques to find ways to create more intelligent agents and that choose their actions through deliberative processes through prior knowledge about the domain that surrounds. Even with the effort of the community to keep this vast area of planning always updated, several works can be revisited and improved, considering that new applications and expansions for the previously used algorithms can be implemented and new problems can be introduced.

In this context, this work seeks to explore action planning for domestic robotics, where ROSPlan, a framework that brings modules for robotic action planning, is used together with Petri nets, to improve the application of both tools together in tasks general purpose in service robotics. In addition, a plan expansion module is proposed, so that we can improve the efficiency of the architecture based on what has already been done by the community previously. To evaluate the proposed model, tests were submitted using a task simulator of the Robocup@Home competition and the performance of the HERA robot in these executions was verified.

Keywords: Probabilistic planning, Service Robotics, Petri Net Plans, Task Planning, ROSPlan

LISTA DE ILUSTRAÇÕES

Ilustração 1 – Representação da interação de um agente genérico com o ambiente.(RUSSELL; NORVIG, 2016) 20
Ilustração 2 – Representação da interação do agente cognitivo com o meio. (RUSSELL; NORVIG, 2016) 22
Ilustração 3 – Grafo direcionado de um problema envolvendo carga e descarga de produtos (GHALLAB; NAU; TRAVERSO, 2004) 25
Ilustração 4 – Representação da expansão dos algoritmos de busca para frente e para trás em um grafo de estados (RUSSELL; NORVIG, 2016) 27
Ilustração 5 – Exemplo de declaração de CPFs, onde as letras p, q e r representam fluentes de estado, e a é uma fluente de ação. (HOLGUIN, 2013) 33
Ilustração 6 – Representação de um plano de execução de tarefa para multi-agentes robóticos (ZIPARO et al., 2011) 36
Ilustração 7 – HERA: Home Environment Robot Assistant (JUNIOR et al., 2019) 37
Ilustração 8 – Funcionamento simplificado da estrutura do *framework* do ROSPlan (CASHMORE et al., 2015) 39
Ilustração 9 – Funcionamento macro da arquitetura proposta e seus módulos individuais 41
Ilustração 10 – Ilustração dos módulos modificados no *framework* do ROSPlan, sendo eles: A base de conhecimento e o despacho das ações 42
Ilustração 11 – Relação semântica entre as palavras do gerador da *Robocup@Home* utilizando a biblioteca SpaCy do Python 44
Ilustração 12 – Resultado do algoritmo que extrai os fluentes da frase de comando para o robô 45
Ilustração 13 – Comunicação dos módulos de entrada da estrutura proposta. 55
Ilustração 14 – Tradução do exemplo "Go to the garage and wait"para uma Petri Nets. . . 55
Ilustração 15 – Trecho de exemplo de um dos comandos em Petri Nets que contém regras de execução, com mais de uma transição podendo ser acionada de um lugar. 56
Ilustração 16 – Representação do mapa, percepção dos sensores e pontos de interesse para a simulação das tarefas. 59
Ilustração 17 – Rede de Petri gerada a partir de experimentos do grupo A de tarefas. . . . 60
Ilustração 18 – Dispersão das médias de tempo obtidas nos experimentos e seus desvios padrão para a primeira rodada de testes. 61
Ilustração 19 – Dispersão das médias de tempo obtidas nos experimentos e seus desvios padrão para a segunda rodada de testes. 61
Ilustração 20 – Relação entre a quantidade de objetos no modelo de mundo e o tempo necessário para produção do plano para a tarefa: "Vá até a dispensa, pegue o refrigerante e coloque-o na mesa da cozinha" 64

LISTA DE TABELAS

Tabela 1 – Dados dos testes realizados em ambiente simulado 60

LISTA DE ALGORITMOS

Algoritmo 1 – Algoritmo para extração dos objetivos do agente a partir de uma frase. . . 45
Algoritmo 2 – Função para aumentar o plano já existente com uma nova consulta ao planejador 57

LISTA DE ABREVIATURAS

ADL	Action Description Language.
CLIPS	C Language Integrated Production System.
HERA	Home Environment Robot Assistant.
IA	Inteligência Artificial.
IPPC	International Probabilistic Planning Competition.
MDP	Decisão de Markov.
PDDL	Planning Domain Definition Language.
PNP	Petri Net Plans.
POMDP	Partially Observable Markov Decision Process.
PPDDL	Probabilistic Planning Domain Definition Language.
PROST	Probabilistic Planning Based on UCT.
RDDL	Relational Dynamic Influence Diagram Language.
ROS	Robot Operating System.
STRIPS	STanford Research Institute Planning System.
UCT	Upper Confidence Bounds Applied to Trees.

SUMÁRIO

1 Objetivo . . . 15
2 Introdução . . . 16
3 Revisão da Literatura . . . 17
4 Conceitos Fundamentais . . . 20
4.1 Agentes e Sistemas Inteligentes . . . 20
4.1.1 Definição de Agente . . . 20
4.1.2 Tipos de Agentes . . . 21
4.2 Planejamento Automático . . . 23
4.2.1 Definição de Planejamento . . . 23
4.2.2 Planejamento Clássico . . . 25
4.2.2.1 Representações . . . 26
4.2.3 Algoritmos de Busca no Espaço de Estados . . . 26
4.2.3.1 Busca para Frente . . . 27
4.2.3.2 Busca para Trás . . . 28
4.2.4 Linguagens de Planejamento . . . 28
4.2.4.1 STRIPS . . . 28
4.2.4.2 PDDL . . . 29
4.3 Planejamento sobre Incertezas . . . 29
4.3.1 Replanejamento . . . 30
4.3.2 Planejamento de Contingência . . . 30
4.3.3 Processo de Decisão de Markov . . . 31
4.3.4 Linguagens de Planejamento . . . 32
4.3.4.1 RDDL . . . 32
4.3.5 Planejador Probabilístico . . . 34
4.3.5.1 PROST-DD . . . 34
4.3.6 PNP - Petri Net Plans . . . 35
4.4 Robótica Móvel . . . 36
4.4.1 HERA - Home Environment Robot Assistant . . . 36
4.4.2 ROS - Robot Operating System . . . 37
4.4.2.1 ROSPlan e ROS_PNP . . . 38
5 Método Proposto . . . 41
5.1 Módulo ROSPlan . . . 41
5.2 Módulo Simulador . . . 42
5.3 Módulo PNP e Executor do Plano . . . 43
6 Desenvolvimento . . . 44
6.1 Tratamento do texto do Gerador de Ações da *Robocup* . . . 44
6.2 Modelo de Mundo em RDDL . . . 45

6.2.1 Arquivo da Instância do Problema 46
6.2.2 Arquivo do Domínio 48
***6.2.2.1 As* pvariables** 49
***6.2.2.2 Descrição das* CPFs *e a* Reward Function** 51
***6.2.2.3 Descrição das* action-preconditions** 53
6.3 Estrutura *ROSPlan* e *Petri Nets* 54
6.3.1 Planejamento das Ações 54
6.3.2 Redes de Petri e Regras de Execução 55
6.3.3 Replanejamento e Agrupamento de Plano 56
7 Resultados 58
8 Conclusão 62
REFERÊNCIAS 66

1 OBJETIVO

Diversas situações no mundo real são imersas em incertezas e um agente robótico necessita trabalhar com o raciocínio sobre seus conhecimentos para criar planos robustos de ações para realizar tarefas complexas em meio a essas incertezas. Para isso, necessitamos de um sistema que atenda aos seguintes requisitos:

- Arquitetura que busca a representação mais realística do ambiente;
- Meios de lidar com eventos não previstos no ambiente, de maneira rápida e evitando replanejamentos;
- Sistema que busca executar objetivos de longo prazo e de recuperação robusta;
- Arquitetura completa e modular, replicável para outros propósitos.

A partir dessas necessidades, o objetivo deste trabalho é a implementação de um planejador não determinístico, ganhador de uma das competições bem conceituadas de planejamento probabilístico em 2018, dentro do *framework* ROSPlan, um dos pacotes usados atualmente para planejamento em robótica, para lidar com as incertezas do ambiente, uma vez que podemos usar probabilidades para descrever o mundo. Não só a integração de um planejador não determinístico, o trabalho se propõe a também integrar no sistema um tradutor dos planos gerados para Redes de Petri, permitindo robustez, concorrência e monitoramento das ações pelo sistema, reduzindo os replanejamentos de planos e "estados mortos"em que o robô não consegue atingir seu objetivo. A arquitetura proposta será testada em um robô móvel em tarefas de propósito geral, onde as regras são aceitas pela comunidade e fornecem um ambiente prefeito para testes, uma vez que a robustez do sistema será posta a prova.

2 INTRODUÇÃO

Sistemas robóticos inteligentes já estão presentes em nosso dia a dia, nos auxiliando e monitorando em diversas atividades graças aos avanços das tecnologias dos sistemas de controle e segurança. Diversos domínios de aplicação já estão sendo colocados em prática, como robôs de assistência social (WILSON et al., 2019; BEMELMANS et al., 2012), robôs de vigilância (DI PAOLA et al., 2010), robôs que fazem visitas guiadas (TOMATIS et al., 2002) ou sistemas de limpeza de piso (KANG et al., 2014) por exemplo.

Nesse cenário, para o domínio de aplicações em ambientes sociais, o agente robótico deve possuir a capacidade de compreender de maneira natural comandos extraídos da interação humano-robô e assim executar as ações necessárias para cumprir o objetivo de maneira geral. Um comando como o "Vá até a cozinha e pegue uma garrafa de água na geladeira"pode ser dado ao robô de serviço e o mesmo precisa raciocinar sobre os passos para que do estado atual possa chegar ao estado final, dando a garrafa que estava na geladeira para o humano por exemplo.

Raciocinar sobre os passos entre uma situação inicial e um objetivo é um típico problema de planejamento de ações. Diversos trabalhos, como em (LIMA; VENTURA; AWAAD, 2018) ou em (CANAL; ALENYÀ; TORRAS, 2019), vêm sendo desenvolvidos rodeando abordagens de planejamento de ações para robôs de serviço, onde modelos de representação do ambiente em bases de conhecimento são usadas para munir o agente de informações e assim algoritmos de planejamento são chamados para com base nas informações da base de conhecimento e do problema proposto, fornecer uma série de pequenas ações que serão transmitidas aos sistemas de baixo nível do robô de modo que ao final o objetivo seja concluído.

Anualmente acontece a competição de robótica RoboCup, onde universidades do mundo todo participam para demonstrar os trabalhos em diversas áreas que são desenvolvidos ao longo de anos em suas instituições. Em 2006 foi criada a liga de robótica doméstica (RoboCup@Home) e desde sua fundação desempenhou um importante papel no intercâmbio de conhecimento de pesquisa e integração entre os integrantes das equipes e quem acompanha o evento, onde os trabalhos desenvolvidos são expostos nos desafios propostos e inovações nas áreas relacionadas são demonstradas nos simpósios. Diversos desafios são colocados em prova pelas equipes e seus robôs, como reconhecimento de pessoas, reconhecimento e síntese de fala, compreensão de linguagem natural, manipulação e reconhecimento de objetos, entre outras tarefas que são exigidas no primeiro estágio da competição em forma de requisição ao robô por um humano. No segundo estágio da competição, os robôs são requisitados por fazerem tarefas que com uma complexidade maior, onde comandos que exigem de uma estrutura de planejamento e monitoramento da execução das ações pelos robôs são colocados em prova, afim de buscar o desenvolvimento dessas habilidades pelas equipes (MATAMOROS; SEIB; PAULUS, 2019).

3 REVISÃO DA LITERATURA

Planejamento de ações em robótica é um assunto amplamente discutido e uma das principais áreas de pesquisa em Inteligência Artificial (IA), para criarmos agentes totalmente inteligentes. Trabalhos que buscam reunir diversos tópicos da engenharia e ciência da computação ao longo de anos para criar sistemas complexos para resolver problemas, como em (SPECK; DORNHEGE; BURGARD, 2017), são os que apresentam maiores desafios e os que buscam maiores avanços para a área de IA, uma vez que consolidam todo o conhecimento já construído em diversas áreas. Dentro do universo de planejamento de ações, o mundo em que o agente se encontra pode ser descrito e compreendido de maneiras simples ou de maior complexidade, que exigem modelos de mundo com maior número de elementos, artifícios de abstração e raciocínio.

Modelos de mundo simples podem ser descritos por representações determinísticas, uma vez que assumimos o mundo totalmente observável e as ações que o agente pode executar têm efeitos que podem ser totalmente determinados, assim trabalhos como (LIMA; VENTURA; AWAAD, 2018), (CROSBY et al., 2016), (MORRIS et al., 2017) e (CASHMORE et al., 2015), demostram a aplicação de planejadores clássicos para resolver problemas em diversos cenários como: robótica de serviço, robôs industriais, veículos aéreos e veículos aquáticos. Os trabalhos tem em comum o uso da linguagem de planejamento PDDL, do inglês *Planning Domain Definition Language*, (YOUNES; LITTMAN, 2004) para descrever as ações, seus efeitos e como o mundo ao redor dos agentes podem ser alterados e de maneira majoritária o *framework* ROSPlan (CASHMORE et al., 2015) é utilizado para controlar toda a arquitetura de planejamento dentro do sistema operacional para robôs (ROS). Outros tipos de planejadores como os temporais, que especificam objetivos que precisam ser alcançados em um tempo finito, foram aplicados em robôs aquáticos descritos em (LUCAS MARTÍNEZ et al., 2020), onde os autores fazem um apanhado de diversas arquiteturas e tipos de planejadores. Para o cenário de planejamento clássico, os possíveis estados de mundo que o agente pode atingir são construídos a partir de proposições sobretudo que o mesmo conhece do mundo (objetos, locais, pessoas, etc) e podemos representar estes por diversas linguagens. Assim, diversos trabalhos também apresentam o uso do PDDL em conjunto com outras descrições de domínio como em (NIEMUELLER; HOFMANN; LAKEMEYER, 2018) que descreve uma arquitetura conjunta com a linguagem CLIPS (*C Language Integrated Production System*) e também trabalhos como (YOUNES; LITTMAN, 2004) que buscam estender o PDDL para incluir desdobramentos probabilísticos no domínio.

Planejadores clássicos determinísticos resolvem diversos problemas, mas no mundo real, em que um agente robótico está imersos, além de fatores que causam efeitos não determinísticos, o mundo pode não ser totalmente observável, trazendo diversas incertezas ao agente na etapa da resolução do problema. A cada fato não esperado no mundo (situação que o robô planeja abrir um porta e a mesma se encontra trancada) o sistema faz uso de estratégias de replanejamento, que busca atrelar o estado atual do agente (com as novas características) ao novo estado inicial e o plano até os objetivos é novamente calculado. Mesmo com estratégias

de replanejamento, as ações replanejadas não garantem que o robô altere o estado do mundo para que consiga atingir o objetivo final, como é mencionado em (LITTLE; THIEBAUX et al., 2007). Assim, trabalhos foram feitos com o propósito de corrigir essas falhas nos planejadores clássicos como em (SANELLI et al., 2017), que introduz o uso de um planejador de contingência (*Contingent-FF*) para gerar planos condicionais e serem executados de maneira controlada por Redes de Petri(RP). Em (LIMA; VENTURA; AWAAD, 2018), (DONDRUP; PAPAIOANNOU; LEMON, 2019), (SHPIEVA; AWAAD, 2015) e em diversos outros trabalhos, planejamento clássico e máquinas de estado finitas são usados para trazer robustez na execução das tarefas, uma vez que os problemas de replanejamento e incertezas sobre o mundo existem e o tempo e o custo computacional podem não ter uma opção para o dinamismo do mundo real.

O planejamento probabilístico é uma abordagem padrão para o planejamento com incerteza em robótica. Uma visão geral das abordagens do planejamento probabilístico é fornecida em (GHALLAB; NAU; TRAVERSO, 2004). A abordagem comum para o planejamento com incertezas em robótica é modelar a tarefa como um Problema de Decisão de Markov (MDP). Em contraste com o planejamento determinístico, onde temos o formalismo em PDDL2.1 (YOUNES; LITTMAN, 2004) usado até agora com o ROSPlan em diversos trabalhos, os cenários de robótica devem frequentemente lidar com eventos não exógenos e incontroláveis (**martinez2017rela+tional**), que podem ser facilmente modelados como POMDP (LITTMAN, 1994). As soluções para os MDPs para robótica já foram exploradas em diversos trabalhos e podem formar políticas com horizonte finito como descrito em (BOUTILIER; DEAN; HANKS, 1999), adotar uma abordagem satisfatória (KUSHMERICK; HANKS; D. S. WELD, 1995) ou maximizar a probabilidade de atingir um estado objetivo (MARTÍNEZ et al., 2017). Em contraste com as linguagens PDDL, e suas extensões, temos o RDDL (SANNER, 2010) que é um formalismo adequado para modelar diversos problemas do tipo MDP e vem sendo usado na IPPC (*International Probabilistic Planning Competition*) como linguagem de entrada para os algoritmos de planejamento mais recentes (GEISSER; SPECK; KELLER, s.d.). É um formalismo dinâmico (Usa as redes Bayesianas), permitindo simultaneidade irrestrita no planejamento das ações.

Esse formalismo é um componente essencial para aplicações de robótica, uma vez que o agente deve executar o planejamento em um ambiente físico com diversas incertezas, como é sumarizado em (MATAMOROS et al., 2018) quando as tarefas de propósito geral, em uma das maiores competições de robótica, começaram a ganhar maior interesse e ser mais desafiadoras. Em (ATRASH; KOENIG, 2001) são observadas que as soluções gráficas de políticas de planejamento dos modelos POMDP, são semelhantes às máquinas de estado finito já descritas em outros trabalhos. Como resultado, ele foi aplicado com sucesso em muitos casos de uso robótico com incerteza, como missões de exploração robótica (SMITH; SIMMONS, 2007).

Em (CANAL et al., 2019) é vista a aplicação do planejador PROST (KELLER; EYERICH, 2012) (campeão por diversas vezes da competição IPPC) para a solução de problemas

não determinísticos em robótica em tarefas de navegação, onde são descritas vantagens no uso desse tipo de planejador em diversas situações, já em (NOGUEIRA, 2018) é vista a aplicação de um sistema que se baseia em um modelo probabilístico em um veículo aéreo, com controle por máquinas de estado para supervisionar a execução das ações para deixar o sistema com um nível maior de robustez e minimizar a quantidade de vezes que o planejador necessita replanejar o plano, mas sem suporte a modelos de concorrência de ações uma vez que a estrutura proposta não obteve sucesso na implementação da estrutura que este trabalho se propõe a implementar (As Petri Net Plans), ambas as soluções embasadas no ROSPlan e com resultados muito positivos em seu uso.

Vimos que diversos trabalhos buscam atingir meios eficientes e robustos de criar agentes que consigam deliberar sobre o ambiente e criar planos para atingir seus objetivos. Para evidenciar os direcionamentos que se seguirão neste trabalho, podemos falar de alguns trabalhos que foram revisados e este trabalho busca seus aprimoramento. Quando falamos de planejamento de ações no *framework* ROS, sem dúvidas o artigo de Cashmore et al. (2015) é o com maior número de citações e possui uma gama de módulos e estruturas já criadas para este tipo de tarefa, a de planejamento. Uma de suas expansões aplica conceitos de planejamento sobre incertezas e faz uso do planejador PROST, descrito em (CANAL et al., 2019), para lidar com ambientes não determinísticos, mas possui alguns problemas, como a limitação dos horizontes do planejador e de ausência de sensibilidade à mudanças no ambiente enquanto executa as ações.

Por fim, além do problema que um planejador resolve ao tentar atingir um objetivo possível, podemos pensar em sistemas que monitoram as execuções das ações vindas do planejador e funcionem para evitar possíveis replanejamentos que possam acontecer. Em (CANAL; ALENYÀ; TORRAS, 2019) podemos ver como um plano pode ser descrito por Redes de Petri e como este sistema pode trazer mais robustez na execução das ações por meio da criação de meios para se recuperar de eventuais problemas no ambiente. Ainda assim, mesmo com esse tipo de tradução, de plano para Redes de Petri, podemos verificar problemas como a necessidade de dinamismo na rede gerada, uma vez que depois da geração a rede criada permanece estática (sem acréscimos de novos lugares nem transições) e a impossibilidade da expansão de novas estruturas da rede através de novas chamadas ao planejador.

4 CONCEITOS FUNDAMENTAIS

4.1 AGENTES E SISTEMAS INTELIGENTES

Com o avanço das tecnologias em diversas áreas como a eletrônica, a mecânica e a computação, o mundo está demandando cada vez mais por sistemas que estejam no meio de diversos ambientes executando parcialmente ou completamente ações que antes eram feitas por humanos. Para tal feito estes sistemas necessitam, além de meios físicos para cumprir com essas atividades, também meios de aprender e compreender os fenômenos concretos e abstratos que ocorrem antes, durante e depois de cada ação que se desenvolve no mundo ao seu redor. Estes sistemas podem conter um ou diversos agentes robóticos, que precisam ser aptos a perceber o ambiente ao redor através de sensores e agir no mesmo através de atuadores.

4.1.1 Definição de Agente

Um agente, seja ele um robô físico que implementa um programa de computador ou um programa de computador puramente dito por exemplo, pode escolher suas ações que irá executar para interagir com seu ambiente dado um histórico de percepções que ele vem acumulando com o tempo até o momento de execução da ação. Em termos matemáticos, pode-se dizer que o comportamento do agente é descrito como uma função que mapeia cada percepção que o mesmo tem e fornece uma ação adequada (RUSSELL; NORVIG, 2016).

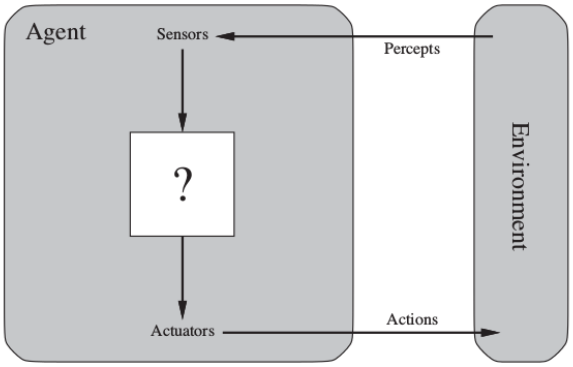

Figura 1 – Representação da interação de um agente genérico com o ambiente.(RUSSELL; NORVIG, 2016)

Com a interpretação do que é um agente, podemos ir além e entender o que para um agente qualquer significa ser racional e as implicações que isso resulta. Ser racional significa fazer a coisa certa em todas as situações, mas o conceito de certo e errado é abstrato e pode ter inúmeras interpretações. Para um agente o conceito de fazer a coisa certa está vinculado a funções que medem sua performance enquanto ele executa suas ações, dando a ele o retorno sobre suas escolhas. O conceito de racionalidade para um agente robótico então, segundo (RUSSELL; NORVIG, 2016), depende de quatro itens:

- A medida de performance para definição de um critério de sucesso;
- O conhecimento prévio do agente sobre o ambiente;
- As ações que o agente pode executar;
- O histórico de percepções que o agente teve até o momento.

Então pode-se definir que para cada possível sequência de percepções o agente racional deve selecionar uma ação que se espera maximizar a performance atual para se atingir um objetivo final com sucesso.

4.1.2 Tipos de Agentes

Dependendo do nível de complexidade exigida pelo ambiente pode-se ter alguns tipos diferentes de agentes para lidar com as situações problema que possam ocorrer. Embora cada tipo possa conter características únicas, um agente inteligente deveria ter três itens: Reatividade (Sendo capaz de perceber e responder de maneira rápida a mudanças), Proatividade (Sendo capaz de, de maneira orientada a um objetivo, tomar uma ação adequada) e Habilidade Social (Sendo capaz de interagir com outros agentes, incluindo humanos, para satisfazer seu objetivo) (WOOLDRIDGE, 2009). Os tipos de agentes, dos mais simples aos mais complexos, são: Agentes Reativos Simples, Agentes Reativos Baseados em um Modelo, Agentes Orientados ao Objetivo, Agentes Baseados em Utilidades e Agentes Cognitivos.

O tipo básico são os chamados agentes reativos. O nome reativo se dá pelo simples motivo de que suas ações são baseadas na percepção instantânea do ambiente, ignorando qualquer acontecimento histórico. As ações deste tipo de agente são totalmente mapeadas e condicionadas à percepção, ou seja, dada uma percepção (o valor da temperatura pelo sensor por exemplo) ele encontra a condição já mapeada para aquela condição e executa a ação (no exemplo da temperatura, quando se elevar a níveis superiores a 35°C, liga-se o ar condicionado). O principal problema envolvendo este tipo de agente é que necessitamos ter um ambiente totalmente observável, a medida que todas as condições possíveis de percepções tem que ser mapeadas e orientadas a alguma(s) ação(ões) (RUSSELL; NORVIG, 2016).

Também um pouco mais simples, mas corrigindo alguns pontos graves no modelo de agente reativo, temos os Agentes Reativos Baseados em um Modelo. Esses agentes lidam melhor com a falta de observabilidade do ambiente, uma vez que um modelo de mundo atualizável é armazenado internamente na sua arquitetura. Essa atualização de estado interno de mundo em sua arquitetura é dependente de dois tipos de conhecimento. O primeiro está associado ao conhecimento que o agente tem do mundo, independente do agente imerso nele e o segundo está associado ao conhecimento de como as ações disponíveis para o agente executar afetam o mundo quando há interação. Esses dois conceitos modelam o que chamamos de modelo de mundo e as ações tomadas agora analisam o estado interno anterior e o como a próxima ação pode afetar o mundo para de fato tomar sua ação (RUSSELL; NORVIG, 2016).

Muitas vezes sabendo sobre o estado do ambiente ou mapeando ações para cada circunstância ainda pode ser ruim para que o agente consiga atingir um objetivo. Faz-se uso então de informações sobre o objetivo do agente incorporadas no modelo interno de sua arquitetura, onde agentes que fazem uso desse tipo de informação são chamados de Agentes Orientados ao Objetivo. Para atingir o objetivo, algumas vezes o agente pode executar uma única ação e outras vezes executar uma sequência de ações, provindas de atividades já mapeadas (Buscas) ou provindas de raciocínio sobre o ambiente e as ações tomadas (planejamento sobre o modelo de mundo existente). Mesmo lentos, esse tipo de agente apresenta ser flexível a medida que para uma nova rota por exemplo, mudamos nosso objetivo, enquanto que no agente do tipo reativo teríamos que escrever uma nova condição que mapeia essa nova rota com a ação que deve ser tomada (RUSSELL; NORVIG, 2016).

Mesmo com os objetivos inclusos no modelo interno dos agentes não há ainda algum meio de saber o grau de sucesso que aquela ação ou conjunto de ações teve ao realizar tal tarefa. Os agentes que trabalham com o conceito de probabilidade de sucesso são chamados de Agentes Baseados em Utilidades, uma vez que analisam a quantidade de sucesso que cada conjunto de ações irá ter, não limitando-se a tratando o objetivo como uma variável binária. Analisar a expectativa de sucesso para cada objetivo é uma das formas para lidar com variáveis reais, onde geralmente não podemos ter todas as condições possíveis de mundo e começamos a trabalhar com o raciocínio sobre as crenças e incertezas. Um agente que possui uma função utilidade explicita pode tomar decisões racionais para problemas de propósito geral e não se limita a tentar maximizar uma função utilidade específica, o que buscamos em arquiteturas de agente complexas.

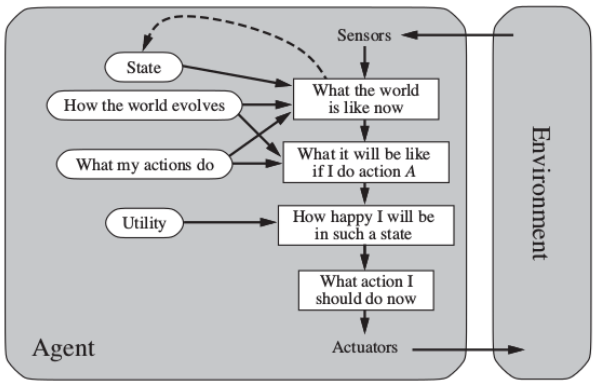

Figura 2 – Representação da interação do agente cognitivo com o meio. (RUSSELL; NORVIG, 2016)

Com um nível de maior complexidade são descritos os Agentes Cognitivos. Esse tipo de agente é desenvolvido para aprender com base em demonstrações ou por suas próprias ações por exemplo. A grande vantagem sobre esses agentes é que eles podem partir de um conhecimento totalmente nulo do ambiente e ir aprendendo com suas experiências até que seu conhecimento futuro seja maior que o conhecimento inicial. O ponto principal da estrutura de um agente cognitivo é a realimentação que ele recebe sobre as ações que ele está executando, a medida que

ele pode se reajustar (usando as medidas de performance) para atingir melhores resultados até seu objetivo bem definido. A Figura 2 apresenta este tipo de agente, onde a partir de interações com o ambiente e a realimentação da percepção, o agente é capaz de construir o raciocínio para melhorar as execuções de ações futuras. Neste tipo de agente, assim como é ilustrado na Figura 2, podemos notar uma estrutura de memória, que lida com a evolução dos estados a cada interação com o ambiente e assim, novas ações são selecionadas a partir do conhecimento prévio dos componentes do mundo e em que estado estes componentes se encontram no momento de execução.

Arquiteturas complexas podem ainda ser definidas com os conceitos já apresentados, uma vez que agentes puramente reativos tem dificuldades em lidar com objetivos e suas ações são regidas à deriva das percepções em tempo atual e agentes puramente deliberativos (que tem meios de raciocinar com base em modelos de mundo) tem problemas de acompanhamento das execução de ações atômicas e sua reatividade a perturbações momentâneas não é muito alta. Diante desses problemas surgem então arquiteturas hibridas, que o foco é mesclar conceitos e funcionalidades de cada tipo de agente para criar uma estrutura em camadas (MÜLLER, 1999).

Com isso são descritas algumas estruturas básicas do funcionamento perceptivo e reativo de agentes, que se estende a diversos tipos de arquiteturas de agentes, como: Agentes Autônomos Genéricos, Agentes de Software, Agentes Inteligentes, Agentes Virtuais, Agentes Móveis, Agentes Informativos e diversos outros. Essas arquiteturas tem uma gama enorme de aplicações, como: Interfaces de sistemas operacionais, processamento de informações, comercio entre outras (D'INVERNO; M. LUCK; M. M. LUCK, 2004).

4.2 PLANEJAMENTO AUTOMÁTICO

4.2.1 Definição de Planejamento

Pode-se definir planejamento como o processo deliberativo que escolhe e organiza ações antecipando seus resultados esperados, ou seja, pré-calculando diversas etapas de um procedimento para se atingir um objetivo, resolvendo o problema. Planejamento automático é uma área de estudo da Inteligência Artificial que estuda esses processos deliberativos (GHALLAB; NAU; TRAVERSO, 2004). No nosso dia a dia lidamos com diversos problemas que muitas das vezes necessitam de planejamento para serem resolvidos, envolvendo situações que nunca vimos antes, atividades complexas de serem executadas ou situações que envolvem ações que não temos muita familiaridade. Em um sistema de planejamento então, a partir de um objetivo bem definido são geradas uma ou diversas ações que possibilitem o agente executor evoluir do estado inicial até seu estado de objetivo final.

Para as diversas formas de ações existentes, também existem diversas formas de planejamentos como: Planejamento de Caminho e Movimento, Planejamento de Percepção, Planejamento de Navegação, Planejamento de Manipulação, Planejamento de Comunicação, entre

outros. Assim, mesmo que existam diversas formas de planejamento o conceito de organização de ações para alterar o estado do sistema é o mesmo e podemos definir um modelo geral para um sistema dinâmico de transição de estados como uma tupla $\sum = (S, A, E, \gamma)$ (GHALLAB; NAU; TRAVERSO, 2004), onde:

- $S = \{s_1, s_2, ...\}$ é um conjunto recursivamente enumerável ou finito de estados;
- $A = \{a_1, a_2, ...\}$ é um conjunto recursivamente enumerável ou finito de ações;
- $E = \{e_1, e_2, ...\}$ é um conjunto recursivamente enumerável ou finito de eventos;
- $\gamma : S \times A \times E \rightarrow 2^S$ é a função de transição de estado.

Podemos ainda representar um sistema de transição de estados por um grafo direcionado, onde os nós contidos neste são os estados no conjunto S de estados possíveis. Se um estado $s\prime \in \gamma(s, u)$, onde u é um par (a, e), $a \in A$ e $e \in E$, então o grafo contém um arco de s (um estado qualquer), até $s\prime$ (um estado futuro ligado pelo arco). Assim o caminho entre um estado inicial e um estado destino é definido por um conjunto de estados intermediários que são ligados por ações e eventos (GHALLAB; NAU; TRAVERSO, 2004).

A figura 3 ilustra um exemplo simples de seis estados de um sistema de carga e descarga. Neste exemplo a problemática envolve a carga de materiais quaisquer por um sistema envolvendo um guincho e um veículo robô. O grafo que representa o sistema é composto por seis estados, seis ações possíveis (Largar, Pegar, Movimento 1, Movimento 2, Carregar e Descarregar) e para as ações de Carregar e Descarregar, por exemplo, podemos pensar em algumas precondições e efeitos que as mesmas têm para com o ambiente. Pensando na **ação de Carga**, a condição para que ela ocorra é o Veículo robô estar disposto no local 1, o veículo robô estar livre para carga e o guincho estar segurando a carga. O efeito imediato para a ação de Carga então é o veículo robô estar carregando o contêiner e o guincho estar livre. Agora para a **ação de Descarga** o cenário da precondição à esta ação é o guincho estar livre, o veículo robô estar o local 1 e o robô estar carregado. Os efeitos causados pela ação de descarga são então, o robô ficar livre e o guincho ficar segurando o carregamento.

Com o exemplo é visto que para este simples grafo podemos descrever os estados que representam o sistema e é possível exemplificar as ações de Carga e Descarga, bem como suas precondições e efeitos que alteram o sistema. Sistemas que envolvem diversas ações e elementos de ambiente desenvolvem estruturas ainda mais complexas e ramificadas, visto que o conjunto de ações que sequenciam os estados até o objetivo é denominado plano. O planejador então é responsável por planejar, dadas as ações e entidades possíveis no sistema, a sequência de ações que irão manipular o ambiente e resultar em um estado final que tem que coincidir com um dos possíveis estados objetivo, onde em nosso exemplo da figura 3 nosso estado inicial poderia ser S_0 e o final S_5.

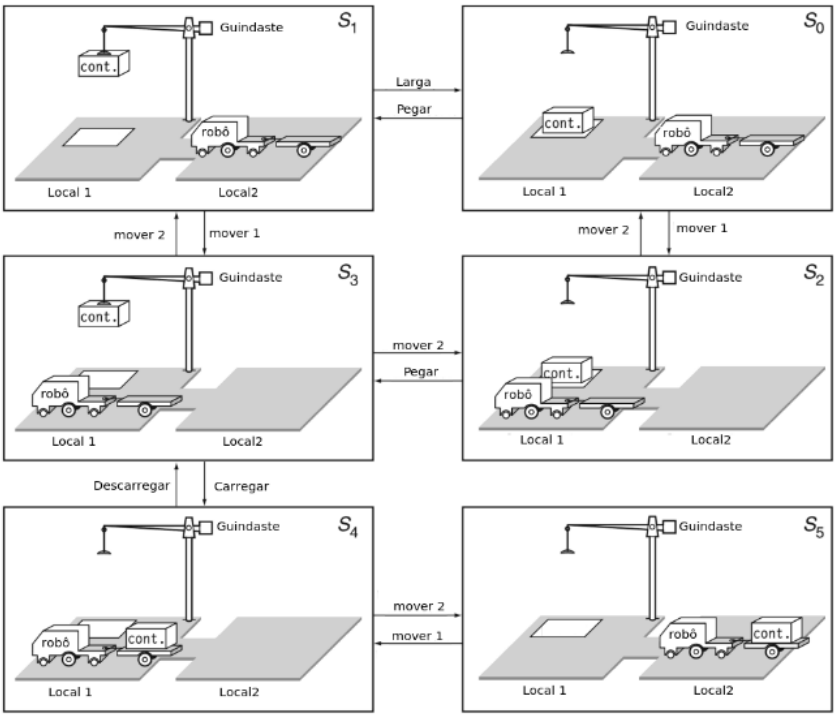

Figura 3 – Grafo direcionado de um problema envolvendo carga e descarga de produtos (GHALLAB; NAU; TRAVERSO, 2004)

.

4.2.2 Planejamento Clássico

É definido o Planejamento Clássico como um sistema de transições de estado restrito, onde é determinístico, estático, finito e totalmente observável. Mesmo com esse sistema sendo tão restrito, pois pode-se entender que no mundo real existem diversos problemas que não são totalmente observáveis ou estáticos por exemplo, é estudado o planejamento clássico para entender como outras formas de planejamento mais complexas são aplicáveis, de modo que na ciência no geral é comum que, quando encontrados problemas de um nível de complexidade elevado, façamos suposições restritas para elaborar modelos bem fundamentados e aproximações.

Um problema de planejamento para um sistema de transições de estado restrito, $\sum = (S, A2, \gamma)$, pode ser definido por $P = (\sum, s_o, g)$, onde temos que s_0 representa o estado inicial do problema e g representa o conjunto de estados objetivos. A solução para P pode ser descrita por um conjunto sequencial de ações $(a_0, a_1, a_2, ..., a_n)$ que correspondem a uma sequência de estados transacionais $(s_0, s_1, s_2, ..., s_k)$, onde s_k é o estado objetivo (GHALLAB; NAU; TRAVERSO, 2004).

4.2.2.1 Representações

Para um problema de planejamento qualquer, é necessário uma descrição do problema em alguma linguagem que os algoritmos de planejamento entendam e resolvam. Para compor uma descrição do problema podemos por exemplo enumerar todos os estados possíveis e todas as ações que fazem a transição destes estados, porém o trabalho descrevendo esse grafo seria maior que solucionar o próprio problema de planejamento em si. Para contornar o trabalho da enumeração de estados e dentre outros benefícios, para o planejamento clássico, três maneiras de representação dos problemas podem ser descritas, de modo que com poucas deduções tenhamos uma abrangência geral do problema, usando axiomas lógicos para inferir coisas ou situações sobre os estados do mundo e fórmulas lógicas para descrever precondições e efeitos das ações de cada problema (GHALLAB; NAU; TRAVERSO, 2004).

O primeiro método de representação é conhecido como Representação Teórica de Conjuntos. Essa forma se baseia na construção de cada estado do mundo por um conjunto de preposições, em que cada ação é uma expressão sintática que especifica quais preposições pertencem ao estado para que a ação seja aplicável e quais preposições a ação adicionará ou removerá para criar um novo estado do mundo.

O segundo método é conhecido por Representação Clássica, onde segue as mesmas abordagens da representação teórica do problema exceto que substitui o uso das preposições, amplamente usadas na representação teórica, por literais de primeira ordem e conectivos lógicos sendo a representação lógica comumente utilizada para sistemas de transição de estados restritos.

Por último temos a Representação de Variável de Estado. Essa forma de representação dos problemas assume que cada estado é representado por um conjunto de valores (conjunto de N variáveis $\{X_0, ..., X_n\}$) e cada ação responsável ro fazer a transição dos estados, é representada por uma função que mapeia o vetor de variáveis em outro vetor de variáveis de saída (próximo estado) (GHALLAB; NAU; TRAVERSO, 2004).

4.2.3 Algoritmos de Busca no Espaço de Estados

O método básico para resolver um problema de planejamento clássico é a busca no espaço de estados. Como pode ser visto no exemplo da figura 4, é possível resumir um problema de planejamento a um problema de busca, onde temos um estado inicial, um grafo de estados disponíveis para acesso e um estado ou conjunto de estados objetivo. Na figura 4, onde as arestas são as ações possíveis (a_n) e os lugares são os estados possíveis no mundo (S_n), vemos a busca para frente sendo representada pela abertura dos estados no espaço, partindo do estado inicial S_0, a cada ação selecionada até que se atinja um objetivo G_0. Na busca para trás o conceito é diferente, a figura 4 apresenta as aberturas dos estados se propagando desde o estado de objetivo G_0 até o estado inicial para cada ação selecionada.

Para resolvermos de maneira simples a busca no espaço de estados proposta, podemos analisar dois algoritmos básicos de busca: Busca para Frente e Busca para Trás; Que serão vistos de maneira aprofundada a seguir e podem ser vistos na imagem.

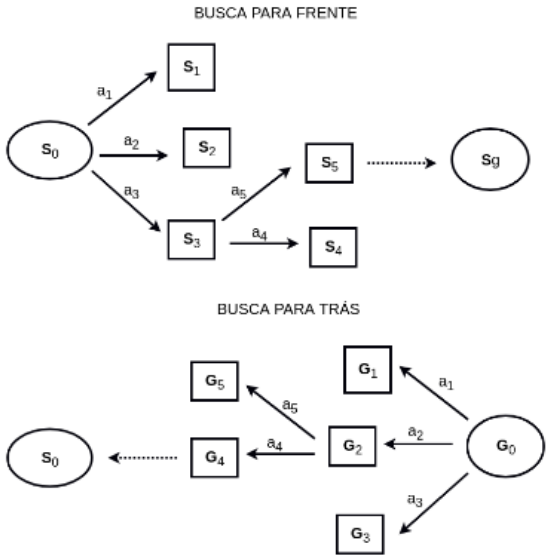

Figura 4 – Representação da expansão dos algoritmos de busca para frente e para trás em um grafo de estados (RUSSELL; NORVIG, 2016)

.

4.2.3.1 Busca para Frente

A busca para frente é um algoritmo de busca que começa pelo estado inicial do problema e vai, de estado em estado, navegando pelo espaço de estados até que atinja o estado objetivo dentre os resultados obtidos. Os estados a frente só são explorados se as precondições das ações que levam a eles são satisfeitas, por exemplo, se a ação $Sobre(A, B)$ tem a precondição da existência de A e B, uma vez que alguma delas não exista a ação não pode ser executada (RUSSELL; NORVIG, 2016).

Um dos problemas da busca para frente é que o algoritmo abre, a cada profundidade de busca, os estados possíveis até encontrar o estado objetivo e isso exige um tempo de processamento que pode ser muito extenso a medida que o problema tem muitas variáveis. Considerando o exemplo onde existam 10 aeroportos e que cada aeroporto possua 5 aviões e 20 peças de carga qualquer, o objetivo aqui é mover toda a carga disponível no aeroporto A para o aeroporto B (RUSSELL; NORVIG, 2016).

Visualizando esse exemplo temos um espaço de estados enorme, dado que cada avião pode voar para algum dos outros 9 aeroportos e cada um dos 200 pacotes de carga pode ser carregado ou descarregado em qualquer avião do aeroporto. Assim, minimamente temos 450 ações possíveis (cada um dos 50 aviões pode voar para algum dos outros 9 aeroportos) de serem aplicadas quando todos os pacotes estão em algum aeroporto e temos 10.450 ações possíveis

de serem aplicadas quando todos os aviões e todos os pacotes estão no mesmo aeroporto. Com isso, assumindo uma média de 2000 ações por estado, o grafo gerado por esse problema tem uma profundidade de cerca de 2000^{41} nós (RUSSELL; NORVIG, 2016).

4.2.3.2 Busca para Trás

A busca para trás tem como encontrar a sequência desejada no espaço de estados (do estado inicial ao estado objetivo) partindo do estado objetivo e retrocedendo os estados até que o estado inicial seja alcançado. Esse método de busca tem algumas vantagens, visto que não é necessário, como a busca para frente, expandir todos os estados anteriores e continuar esse laço de expansão, a medida que vamos verificando as literais de cada precondição das ações anteriores (RUSSELL; NORVIG, 2016).

Esse método, que é chamado também de busca por estados relevantes, abre os estados pela regressão apenas quando as ações que levaram a ele são relevantes para a obtenção do meu objetivo. Quando temos, por exemplo, o objetivo $\neg Pobre \wedge Famoso$, entendemos que Pobre é falso, Famoso é verdadeiro e qualquer outro fluente pode adotar qualquer valor. Quando o algoritmo é executado, ele irá verificar se os efeitos da ação se unem ao objetivo e não negue nenhum fluente (não cause $\neg Famoso$ por exemplo), tornando a ação relevante para o meu problema (ajudando a chegar mais perto do objetivo) e assim por diante (RUSSELL; NORVIG, 2016).

4.2.4 Linguagens de Planejamento

4.2.4.1 STRIPS

A sigla da linguagem STRIPS representa *STanford Research Institute Planning System* (FIKES; NILSSON, 1971). É uma linguagem formal usada para descrever domínios determinísticos, estáticos e totalmente observáveis. Neste formalismo, assume-se que o estado inicial do mundo é totalmente conhecido e qualquer ação tomada em qualquer estado do sistema tem no máximo um sucesso (ou estado resultante). As ações não possuem descrições sobre seus efeitos indiretos e as precondições de cada uma delas são descritas simplesmente por conjuntos de literais. Uma instância da linguagem STRIPS é representada por uma tupla (P, O, I, G), onde P é um conjunto de condições proposicionais, O é um conjunto de operadores ou ações, I é o estado inicial e G é o estado final esperado.

Um planejamento gerado pela STRIPS é representado por uma sequência linear de ações do estado inicial ao estado final sem ramificações (causadas por efeitos de ações), o que causa a obtenção dos planos bem rápido, sendo possível seu processamento *off-line* sem problemas computacionais.

Como o modelo gerado pela linguagem é representado sem efeitos condicionais de ações, o plano pode aumentar de tamanho e exigir grande processamento computacional (para modelos complexos), assim uma versão estendida da STRIPS, a *Action Description Language (ADL)*, foi criada (PEDNAULT, 1987). Assim a ADL inclui na STRIPS: Possibilidade de inclusão de literais negativas, adição de efeitos condicionais e operadores lógicos para expressá-los e disjunções (ampliando as especificações dos estados de objetivo).

4.2.4.2 PDDL

A PDDL é uma linguagem padrão para definição de domínios de planejamento que surgiu para generalizar a linguagem STRIPS e ADL (AERONAUTIQUES et al., 1998). A linguagem PDDL, criada inicialmente para a competição IPC (*International Planning Competition*) que ocorreu na conferência AIPS-98 (*Artificial Intelligence Planning System*) para padronizar domínios de planejamento, descreve o domínio que o agente irá interagir e o próprio agente com base em termos de predicados, ações possíveis para o domínio e os efeitos que essas ações tem no ambiente e no agente.

Os estados em PDDL são representados como um conjunto de predicados e são usadas sentenças de lógica de primeira ordem para descrevê-los. As ações produzem efeitos no mundo e estes são a única forma de se alterar o estado de mundo para o domínio em PDDL, podendo ser quantificados e condicionais. A versão básica da PDDL divide basicamente um problema de planejamento em duas partes: A descrição do domínio, que contem todos os elementos do mundo que são comuns a cada problema do domínio e a Descrição do problema, que contém os elementos que são específicos para aquela instância do problema (cada vez que um novo problema é requisitado ao planejados, uma nova descrição do problema é criada, mas a descrição do domínio é a mesma). Essas duas divisões para um problema de planejamento são separadas em dois arquivos para a PDDL (NOGUEIRA, 2018).

4.3 PLANEJAMENTO SOBRE INCERTEZAS

Quando lidamos com problemas no mundo real nosso contexto de aplicação é repleto de incertezas. Esses problemas muitas vezes podem ser aproximados por problemas determinísticos e que todos os estados são observáveis, assim é possível resolvê-los com os métodos clássicos, mas muitas vezes os métodos de solução não apresentam meios de se chegar ao objetivo com essas aproximações. Um exemplo para as incertezas do ambiente é um táxi indo de um ponto A a um ponto B, onde desenvolve uma sequência de ações que o permite concluir a rota, contudo nessa aproximação determinística é levado em consideração o táxi quebrar, o trânsito, desvios de rota e outras incertezas que são notadas pelo ambiente e pelo próprio agente.

Para estender os meios determinísticos para se planejar uma solução para um problema real, esta secção irá se aprofundar por métodos que são utilizados para lidar com essas incer-

tezas do ambiente, como o replanejamento das ações, planos de contingência e por fim sobre probabilidades associadas ao raciocínio do agente para lidar com essas situações.

4.3.1 Replanejamento

Quando algo inesperado acontece na execução dos planos o agente precisa replanejar suas ações para que atinja o objetivo, de modo que o replanejamento é um evento que acontece *online*. O evento de replanejar, que acontece então quando um agente se depara com um efeito não previsível de suas ações, usa o estado atual que o agente se encontra e aproxima todas suas observações não determinísticas por determinísticas, atualizando sua base de conhecimento e por fim traçando novas ações para alcançar o objetivo (HARRIS, 2015). Com tudo, um problema que temos no evento de replanejamento é que ele não raciocina sobre os diferentes cenários possíveis e duas situações são críticas para o sistema e sua recuperação: os estados de bloqueio mortal (*dead-ends*) e os bloqueios com interações repetitivas (LITTLE; THIEBAUX et al., 2007).

Contudo, um planejador simples que utilizou estratégias de replanejamento para lidar com as incertezas do ambiente foi o FF-Replan que, como descrito em (HARRIS, 2015), mesmo não oferecendo garantias para evasão de estados de bloqueio mortal superou diversos planejadores probabilísticos e venceu a primeira IPPC em 2004 e com algumas modificações, excedeu todos os participantes da IPPC em 2006 com algumas modificações.

4.3.2 Planejamento de Contingência

Um outro meio de lidar com as incertezas sobre o mundo são os planos de contingência. Planejamento de contingência é a tarefa de gerar planos condicionais dadas as incertezas sobre o estado inicial do agente ou sobre os efeitos de suas ações ao longo da execução dos planos (HOFFMANN; BRAFMAN, 2005). Esses planos condicionais são gerados uma vez que o planejador de contingência pode lidar com literais indeterminadas (variáveis que não são determinadas em tempo de planejamento) e assim, para cada indeterminação o planejador constrói uma ramificação condicional no plano e deixa a tomada de decisão (observabilidade da literal e qual rumo tomar da ramificação) em tempo de execução. Em robótica de serviço podemos exemplificar a literal desconhecida como ter uma pessoa em um certo cômodo, onde não temos certeza sobre essa informação e o planejador pode criar um plano condicional para avaliar outro cômodo dado que a pessoa não esta lá Destacando para o exemplo que o agente limita-se a ter a informação pelos sensores se a pessoa está la ou não em tempo de execução, com o agente já no cômodo.

Um dos algoritmos usados para problemas dessa natureza é o Contingent-FF, que foi adaptado em (HOFFMANN; BRAFMAN, 2005) de planejamento conformante, uma vez que problemas conformantes são aqueles que possuem diferentes situações possíveis e que possuem

uma solução capaz de alcançar os objetivos com uma sequencia de ações, independente das incertezas. Um exemplo da aplicação do Contingent-FF em robótica de serviço é mostrado em (SANELLI et al., 2017), onde as incertezas do ambiente são simplesmente dribladas por planos condicionais criados dentro do *framework* ROSPlan.

Uma desvantagem da representação dos efeitos não-determinísticos da ação por observabilidade parcial é que o agente é incapaz de observar o resultado de uma ação várias vezes para assegurar seu valor, tratando seus efeitos como eventos independentes no tempo. Assim, para as situações de contingência, os mecanismos dos agentes para recuperação após uma emergência e sua percepção ficam limitados.

4.3.3 Processo de Decisão de Markov

Um processo de decisão de Markov é uma representação estocástica do estado de um sistema dinâmico, no qual toda transição de estados contém uma probabilidade associada. Falamos que um processo é "Markoviano"quando o estado futuro depende apenas da ação tomada e do estado atual, não sendo considerados os estados anteriores. A escolha das ações a serem tomadas pelo agente em um processo Markoviano são especificadas por uma função utilidade, que orienta o agente através da meta. Assim, segundo (RUSSELL; NORVIG, 2016), podemos definir um processo Markoviano como uma 5-tupla (S, A, T, R, γ) e temos que:

- $S = \{s_1, s_2, ...\}$ é um conjunto de estados finitos;
- $A = \{a_1, a_2, ...\}$ é um conjunto de ações finito;
- $T : S \times A \times S \rightarrow [0, 1]$ é uma função de probabilidade de ocorrência de um estado $s' \in S$ ao executar uma ação $a \in A$ em um estado $s \in S$, ou seja, $T(s'|s,a)$;
- $R : S \times A \rightarrow IR$ é uma função que relaciona uma recompensa ou custo à aplicação de uma ação $a \in A$ no estado $s \in S$, $R(s,a)$;
- $\gamma \rightarrow [0,1]$ é o desconto dado à uma recompensa futura.

A solução para resolver problemas deste tipo, consiste em encontrar uma $política \prod$ que maximize a recompensa média do agente determinando a melhor ação a ser tomada em um dado estado s, sendo assim a política relaciona um estado à uma ação: $\prod : S \rightarrow A$. Um processo Markoviano permite que partindo de qualquer estado seja possível atingir o objetivo, sempre questionando a política, e tradicionalmente a solução é obtida de maneira iterativa por valor e por política, baseando-se na equação de *Bellman*:

$$V^{\pi}_{i+1}(s) = max_a\{\sum_{s'} T(s'|s,a)(R(s,a) + \gamma V^{\pi}_i(s'))\} \tag{1}$$

As iterações para a resolução do problema, descritas pela equação de *Bellman*, buscam associar recompensas aos estados objetivos e recompensas nulas para os demais, de modo que à medida que as iterações avançam os valores $i+1$ são resolvidos com base nos valores da iteração anterior. A solução de uma MDP pode ser adquirida de duas formas com essa equação: Com

horizontes finitos, que são determinados o número de iterações e, Com horizontes infinitos, que é definido um critério de parada. Na solução da equação por iteração de valor,o critério de parada é estabelecido à medida que ocorre a convergência dos valores de recompensa de todos os estados, visto que em grande parte destes problemas este valor cresce indeterminadamente conforme o número de iterações e a convergência é garantida então com a introdução de um fator ($\gamma \rightarrow [0,1]$) que deprecia as recompensas de estados futuros. Na iteração por política é realizada a iteração de valores monitorando-se a política a cada iteração $\pi(i+1)$, sendo que o critério de parada ocorre quando $\pi(i) = \pi(i+1)$.

A resolução de uma MDP, apresentada por exemplo pela iteração da equação de *Bellman*, é só um dos exemplos clássicos para se resolver problemas deste tipo. Nas competições IPPC são apresentados diversos outros algoritmos estado da arte que solucionam os problemas na categoria MDP, como é o caso da técnica de estado da arte PROST.

4.3.4 Linguagens de Planejamento

4.3.4.1 RDDL

A linguagem RDDL nasceu com o intuito de modelar problemas que não podiam ser tratados em outras linguagens de planejamento já existentes (PPDDL por exemplo) e foi primeiramente proposta para resolver estes problemas na IPCC (SANNER, 2010). Problemas complexos de planejamento exigem uma descrição com maior detalhamento sobre o ambiente e seu comportamento, não podendo ser descrito simplesmente com os conceitos usados na linguagem PPDDL, uma vez que ela não permite representar tópicos como: ações concorrentes; múltiplos eventos exógenos que afetam o sistema; diferentes distribuições de probabilidades nas transições; observação parcial(estados de crença); a diferença entre fluentes e não-fluentes; a função recompensa e a probabilidade de transição através de funções não-lineares, etc (HOLGUIN, 2013).

Na linguagem RDDL todas as variáveis são parametrizadas (as chamadas pvariable, dentro da linguagem), incluindo as variáveis dos tipos fluentes de ações, de estados, de observações, de intermediários e as do tipo não-fluente (fluentes que tem seu valor como uma constante). A semântica da linguagem é a mesma de uma Rede Bayesiana Dinâmica estendida a um diagrama de influência, com estados fatorados, observações fatoradas, ações fatoradas e concorrentes, fluentes de estados intermediários (predicados derivados, efeitos correlacionados, estratificação por níveis que define um DBN relacional com múltiplas camadas) e os eventos exógenos independentes (HOLGUIN, 2013).

Em RDDL, devemos especificar Funções Condicionais para todas as pvariables declaradas no contexto do problema. Essas funções, que podem ser unções condicionais probabilísticas ou funções condicionais determinística, ditam o que acontece com cada variavel no sistema a

cada nova iteração. Essas Funções Condicionais Probabilística podem ser compostas por (HOLGUIN, 2013):

- constantes que podem assumir valores como: true, false, inteiros, reais, valores enumerados;
- agrupamentos, através de parenteses ou colchetes;
- expressões lógicas;
- expressões aritméticas;
- expressões de comparações com igualdade ou desigualdade;
- expressões condicionais if-then-else e switch;
- distribuições de probabilidade básicas.

A partir disso, toda a lógica em RDDL é construída dentro das funções condicionais, onde devemos apontar o que acontece com o próximo estado de cada uma das variáveis, como é mostrado como exemplo de declaração a figura 5. Na figura as variáveis de estado p, q e r tem seu funcionamento descrito por um fator condicional e seu valor é assumido por uma distribuição. Usando como exemplo a variável p, o p' representa o valor da variável num próximo instante e a cada instante todas as condições são verificadas e os valores são assumidos para todas as variáveis (HOLGUIN, 2013).

```
cpfs {
    p' = if (p ^ r)
         then Bernoulli (.9)
         else Bernoulli (.3);
    q' = if (q ^ r)
         then Bernoulli (.9)
         else if (a)
              then Bernoulli (.3)
              else Bernoulli (.7);
    r' = if (~ q)
         then KronDelta (r)
         else KronDelta (r <=> q);
};
```

Figura 5 – Exemplo de declaração de CPFs, onde as letras p, q e r representam fluentes de estado, e a é uma fluente de ação. (HOLGUIN, 2013)

.

A separação estrutural desta linguagem se assemelha a outras de mesmo tipo, como a PDDL ou PPDDL, sendo que um arquivo da estrutura é destinado a descrição do domínio (descrição de requisitos do domínio, definição dos tipos de parâmetros, definição das variáveis de domínio, definição das funções de transição probabilísticas e da função recompensa do problema) e outro é destinado às instâncias do problema (descrição do estado inicial, dos objetos usados e um critério de meta).

4.3.5 Planejador Probabilístico

4.3.5.1 PROST-DD

O algoritmo PROST-DD (GEISSER; SPECK, 2018), ganhador da IPPC de 2018, se assemelha a seu predecessor PROST (KELLER; EYERICH, 2012). Ambos algoritmos são baseados em planejamento UCT (Upper Confidence Bounds applied to trees) (KOCSIS; SZEPESVÁRI, 2006), que trata o problema da exploração versus explotação, onde o agente tenta maximizar a recompensa obtida explorando novas ações para atingir seu objetivo final ou desenvolvendo as mesmas com a mesma finalidade. Com isso, quando o agente está explorando, ele esta intrinsecamente testando ações diferentes para atingir um objetivo e quando o mesmo está explotando, ele efetua simulações de Monte-Carlo para uma dada ação. O UCT constrói uma árvore do sistema, sendo que em cada nó são armazenados um estado e um contador de visitas para esse estado. O UCT é um algoritmo de planejamento baseado em técnicas de amostragem de Monte-Carlo projetado para resolver MDPs e ao contrário de outras técnicas com o mesmo propósito, o UCT precisa fazer amostragem da função de transição e da função recompensa de um simulador, não precisando conhecer as probabilidades de transição ou valores da recompensa das ações. O UCT realiza um número de testes, chamados de *rollouts*, através do espaço de estados e atualizações da função valor nos estados que visita nessas trajetórias. A escolha dos estados sucessores é estocástica, porém a escolha das ações é gulosa. Os testes que o algoritmo realiza, sempre começam da raiz da árvore e acabam num estado objetivo ou em um estado que não faz parte do grafo. A recompensa quando um estado novo é gerado, é amostrada a partir de uma simulação por uma política base do estado inicial, assim os valores são propagados usando as atualizações de Monte-Carlo.

Com isso, o planejador Prost (KELLER; EYERICH, 2012) usa o algoritmo UCT, descrito a cima, e introduz algumas otimizações para melhorar seu desempenho, entre elas, técnicas para reduzir o fator de ramificação e para limitar a profundidade da busca aplicadas no grafo UTC criado. O algoritmo PROST-DD, que também aplica o algoritmo UTC, apresenta uma nova heurística para inicializar novos nós de decisão da arvore criada pelo seu processamento interno. O *framework Trial-based Heuristic Tree Search* (THTS) realiza a criação e expansão de uma árvore de busca para a resolução do problema e a função heurística do algoritmo PROST-DD se aplica quando, dado um estado qualquer, quando uma ação pode ser tomada o algoritmo expande o nó e atribui um valor de estado à ele. Esse valor de estado atribuido é baseado na média dos ganhos da expansão dos seus nós sucessores. De maneira geral, o algoritmo se mostrou eficiente para tratar de problemas complexos e com tempo de execução menor, em comparação à outros da mesma categoria (GEISSER; SPECK, 2018).

4.3.6 PNP - Petri Net Plans

Podemos descrever as *Petri Net Plans (PNP)* como um formalismo que pode representar planos complexos (um conjunto de ações que interagem de formas complexas por exemplo) fornecendo modelos condicionais para os planos descritos por ele. As *PNPs* são úteis para sistemas não determinísticos, ou seja, nem todos os estados são completamente observáveis e podem ser aplicados a diversos problemas, como: robôs sociais, sistemas multi-agente, agentes de videogames, interações humano-robô, etc.

Os *PNPs* tem maior expressividade que as Máquinas de Estado Finito (FSM) e permitem a análise automática do plano, que pode fornecer garantias formais sobre o desempenho dos planos gerados. Depois de gerada, a execução das *PNPs* é extremamente eficiente e permite o desenho de comportamentos pró-ativos e reativos em tempo real (ZIPARO et al., 2011). Exemplos do uso com sucesso das *PNPs* podem ser vistos em (BASTIANELLI et al., 2013), (FARINELLI et al., 2006) e (PALAMARA et al., 2008).

Um plano gerado por esse formalismo é representado por uma Rede de Petri aumentada com um conjunto de marcações de objetivos G. Formalmente, é uma tupla *(P, T, F, W, M0, G)* em que:

- $P = \{p_0, ..., p_n\}$ é um conjunto finito de locais;
- $T = \{t_0, ..., t_{n]}\}$ é um conjunto finito de transições;
- $F \subseteq (P \times T) \cup (T \times P)$ é um conjunto de arestas;
- $W : F \rightarrow 1, 2, 3, ...$ é uma função de ponderação;
- $M_0 : P \rightarrow 0, 1, 2, ...$ é a marcação inicial
- $P \cup T \neq 0$ e $P \cap T = 0$.

Dizemos então que as redes de Petri representam um sistema por um grafo que tem como características ser direcionado, bipartido e ponderado e pode ser construído com duas estruturas básicas: Os locais e as transições. Os locais representam as fases de execução de uma ação, enquanto transições representam eventos. Em particular, cada ação é representada por três estados: inicial, execução e estado final. Entre eles, há a ação que inicia a transição, a ação que termina a transição e pode haver alguma ação que interrompe e/ou controla as transições.

A evolução da Rede de Petri é descrita pela regra de disparo: uma transição é ativada quando o número de marcadores para seus locais iniciais for igual pelo menos ao número de marcadores do peso da borda. Quando a transição é acionada, um número de marcadores iguais o peso das arestas de entrada é removido dos locais iniciais e um número de marcadores iguais às bordas de saída são adicionados aos nós de saída. Os planos da Rede Petri são definidos em termos de ações e operadores. A figura 6 mostra um exemplo de plano complexo gerado em Redes de Petri para um sistema multi-robô. Na figura são representados os lugares da rede, sendo as ações em execução pelo agente, as transições, que representam os gatilhos de início e fim de cada ação a ser executada pelo agente e por fim as marcas, que representam qual ação está em execução pelo agente em cada instante de tempo até o lugar que representa o objetivo.

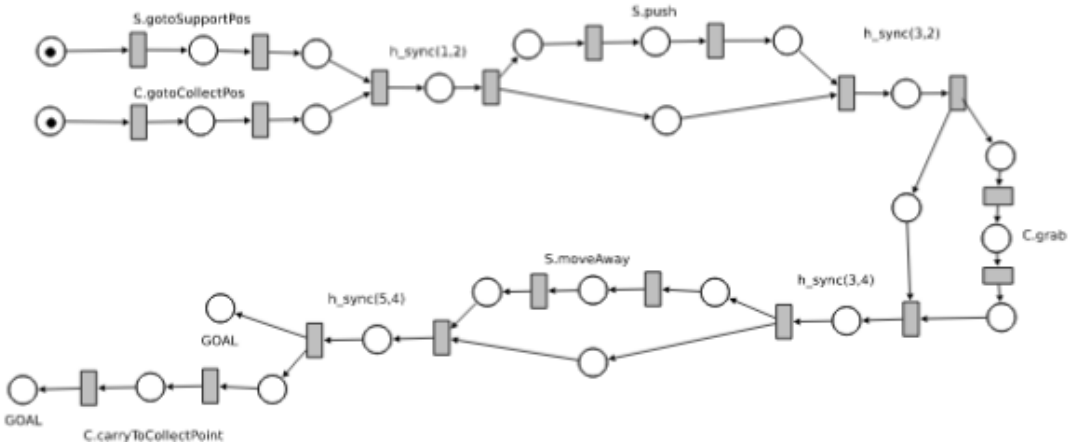

Figura 6 – Representação de um plano de execução de tarefa para multi-agentes robóticos (ZIPARO et al., 2011)

.

4.4 ROBÓTICA MÓVEL

Como descrito anteriormente, agentes inteligentes são capazes de resolver problemas dos mais diversos possíveis. Quando falamos de problemas no contexto doméstico, hospitalar, comércios ou fábricas, necessitamos destes agentes robóticos, autônomos na maioria das vezes, para interagir com estes ambientes complexos. A situação a ser solucionada pelo agente é que diz o quão robusto ou equipado estes deve ser para conseguir realizar as atividades no domínio sem nenhum problema. Exemplos destes são robôs usados para pegar objetos, que necessitam de manipuladores, ou até mesmo robôs que guiam pessoas por um hospital, sendo necessário o uso de rodas e das voz, assim no âmbito da robótica móvel, vemos agentes robóticos que podem se deslocar pelo domínio para realizar seus fins.

Quando falamos de sistemas robóticos móveis, contamos com uma série de módulos operacionais que o robô necessita para estar em pleno funcionamento: Módulos elétricos, módulos mecânicos e módulos de processamento computacional. A medida que o robô autônomo faz uso destes módulos, sendo com maior ou menos complexidade, são necessárias ferramentas para controlar e organizar as tarefas de cada um de forma hierárquica, assim diversas linguagens de programação podem ser utilizadas, sendo de maior ou de menor nível, para que o agente possa controlar e executar suas tarefas de maior nível: pegar um objeto em cima da mesa ou se deslocar até um ponto em específico de um ambiente.

4.4.1 HERA - Home Environment Robot Assistant

O robô HERA (JUNIOR et al., 2019), em sua versão desde 2019, é um projeto de robótica em andamento desenvolvido pela equipe RoboFEI@Home da faculdade Centro Universitário da FEI. O projeto é focado em desenvolver pesquisa nas áreas de interação humano-

máquina, contribuindo assim nos avanços do estado da arte e aplicações envolvendo interfaces adaptativas, interfaces cérebro-computador, planejamento de ações, automação de ambientes residenciais inteligentes e internet das coisas.

O robô possui diversos elementos que o ajudam na execução das tarefas propostas em ambientes domésticos e assim, tanto o *software* quando o *hardware* contribuem em módulos principais do robô como: Visão Computacional, Reconhecimento de Fala e Navegação. Cada um destes módulos possui interface com diversos sensores que trocam informações ente si em um sistema embarcado acoplado em um computador central (JUNIOR et al., 2019). A figura 7 ilustra a estrutura mecânica do robô com todo o conjunto de sensores, atuadores e as unidades de controle.

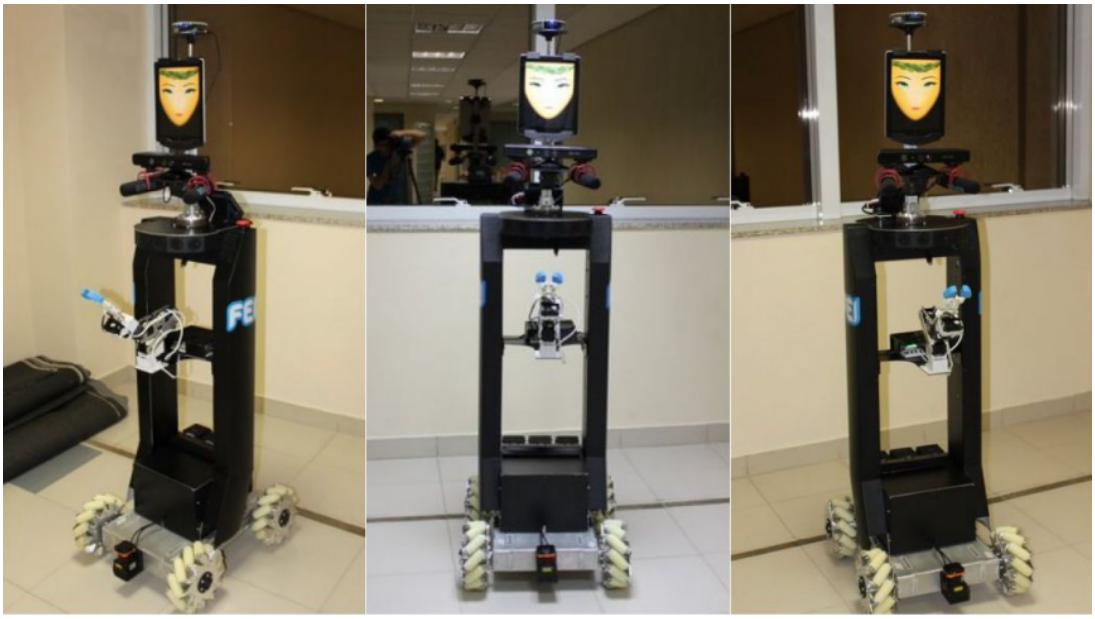

Figura 7 – HERA: Home Environment Robot Assistant (JUNIOR et al., 2019)

.

4.4.2 ROS - Robot Operating System

Robot Operating System ou Sistema Operacional de Robôs é uma plataforma com um conjunto de ferramentas para desenvolvimento de softwares em robótica, ele fornece abstração de hardware, visualizadores, transmissão de mensagens e serviços, um conjunto amplo de bibliotecas, gerenciamento de pacotes e diversas outras ferramentas. Um dos fatores importantes sobre seu uso é sobre sua licença, que é totalmente *Open Source*, ou seja, tem seu código aberto e possibilita sua total modificação e melhora por outros desenvolvedores.

O ROS tem uma comunidade grande, visto que, a maioria dos pacotes se mantém atualizado e/ou outras versões são lançadas por outros membros da comunidade a medida que a contribuição científica/tecnológica está sempre se renovando. Seu funcionamento está segmentado em *packages* ou pacotes, que juntos trazem toda a funcionalidade para a plataforma. Sua

estrutura principal é formada por *Stacks* que são as unidades de maior nível no ROS, (um exemplo de *Stack* seria a "navegação" do robô autônomo) dentro dos *Stacks* podem ter um ou vários pacotes, onde todos os pacotes dentro de um *Stack* estão associados à mesma funcionalidade (no *Stack* da navegação, por exemplo, existem pacotes ligados à como a navegação do robô irá funcionar, como pacotes que leem sensores, pacotes que planejam rotas, entre outros). Dentro dos pacotes existem os códigos ou algoritmos que fazem aquele pacote desempenhar sua função, sendo que, dentro de um pacote podem ter um ou vários *scripts* ou códigos (por exemplo, no pacote que lê dados de um laser para enviar para o sistema, podem ter códigos que enviam o sinal do laser para a plataforma, para habilitá-lo ou desabilitá-lo, para verificar se a comunicação do hardware com o software está em plenas condições, entre outros códigos que dizem respeito à leitura das informações).

Uma das coisas atrativas nessa plataforma é sua facilidade de implementação e flexibilidade de hardware. Imagine que existem diversos tipos de plataformas mecânicas na robótica, manipuladores, robôs móveis de diversos modelos, robôs aéreos (como drones, aviões autônomos com sistema embarcado, entre outros), ou qualquer outro tipo de hardware que tenha um software embarcado voltado à robótica, agora imagine todo o trabalho de se construir praticamente do nada todo o código de controle para cada nova plataforma dessa. O ROS veio para melhorar esta situação, à medida que com sua utilização embarcada no software dos robôs com os requisitos mínimos, torna o código que foi escrito para um robô aéreo por exemplo, ser facilmente modificado, em algumas variáveis, para um robô terrestre e assim por diante. Com essa versatilidade não é necessário começar um projeto do zero, na visão do software embarcado, toda vez que um novo modelo de plataforma mecânica é desenvolvido, assim a pesquisa pode ser reproduzida facilmente por diversos pesquisadores e empresas sem grandes problemas (QUIGLEY et al., 2009).

4.4.2.1 ROSPlan e ROS_PNP

Para trabalharmos com sistemas de planejamento dentro do *framework* ROS, (CASHMORE et al., 2015) propôs uma biblioteca que integra os recursos já existentes no ROS (como execução e monitoramento de ações, módulos de leitura de sensores, etc) com planejadores clássicos genéricos. Toda a arquitetura da biblioteca, junto ao sistema ROS, é altamente modular e fornece ferramentas para:

- A partir dos dados obtidos de sensores do robô, gerar automaticamente o estado inicial lógico em PDDL;
- Executar de forma automática as chamadas de execução do planejador de ações;
- Executar o processamento e a validação do plano gerado;
- Manejar o módulo de despacho de ações planejadas, de modo à monitorar falhas nas ações e mudanças de ambientes;

- Lidar com a tradução das ações geradas pelo planejador em ações dentro da estrutura do ROS.

A estrutura principal de funcionamento do ROSPlan está associada a dois nós dentro do *framework* do ROS, onde um deles é responsável por administrar a base de conhecimento (KB) (atualizando valores lógicos a partir de dados provindos de sensores) e o outro executa o planejamento do sistema (PS). A figura 8 ilustra o funcionamento dos nós descritos, bem como os dados provindos da estrutura do ROS e o fluxo de execução no ambiente. Basicamente, as informações que os sensores adquirem do ambiente são tratadas por módulos individuais e introduzidas no ROSPlan, que irá utilizar estes dados para instanciar os problemas de planejamento e realizar o envio do plano contendo as ações a serem executadas como mover um braço robótico, mudar de ambiente ou capturar uma imagem do sistema de visão por exemplo.

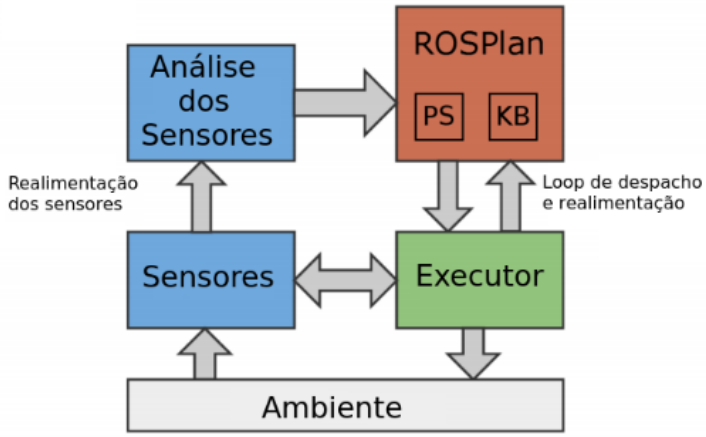

Figura 8 – Funcionamento simplificado da estrutura do *framework* do ROSPlan (CASHMORE et al., 2015)

.

A base de conhecimento, a sigla KB na figura 8, está associada à obtenção das informações de diferentes fontes, como: Informações de sensores, informações descritas pelo usuário, informações adquiridas do domínio e informações adquiridas durante a execução do plano. Essas informações adquiridas e formatadas na base de conhecimento serão utilizadas no nó de planejamento para gerar o arquivo problema na linguagem PDDL, transformar as ações geradas pelo sistema de planejamento em ações executáveis pelo ROS e monitorar o estado do ambiente e notificar o ROS caso as mudanças invalidem o plano em execução.

Durante a execução do plano gerado pelo *framework* pode ser necessário o replanejamento das ações, que é baseado na reformulação do problema, e pode se dar por três razões: a ação que está sendo executada retornou uma falha; a base de conhecimento informa ao planejador uma alteração que invalida o plano ou aparece uma nova informação importante para o plano; ou quando a ação atual supera significativamente o tempo planejado. No primeiro caso dos citados acima, como a repetição de despacho da ação já está quebrado, é necessário, apenas, que seja gerado uma nova instância do problema a partir do modelo atual do ambiente e depois

construir um novo plano. Já nos casos 2 e 3 apontados no tópicos acima é necessário primeiro parar a execução do plano para depois começar o replanejamento.

5 MÉTODO PROPOSTO

Todo *framework* proposto tem seu funcionamento em módulos, onde cada um tem um papel desde a compreensão dos problemas que serão resolvidos pelo agente robótico até a execução dos planos gerados. A figura 9 ilustra os blocos principais do sistema e como eles se interagem, onde a partir dos dados de sensores, do problema e da base de conhecimento, é gerada uma política para o bloco que simulara e cria o plano inicial a ser executado pelo agente. Após a criação do plano inicial, o módulo PNP irá construir a rede de Petri e controlará as execuções, sendo que toda essa arquitetura pode ser vista na imagem a seguir.

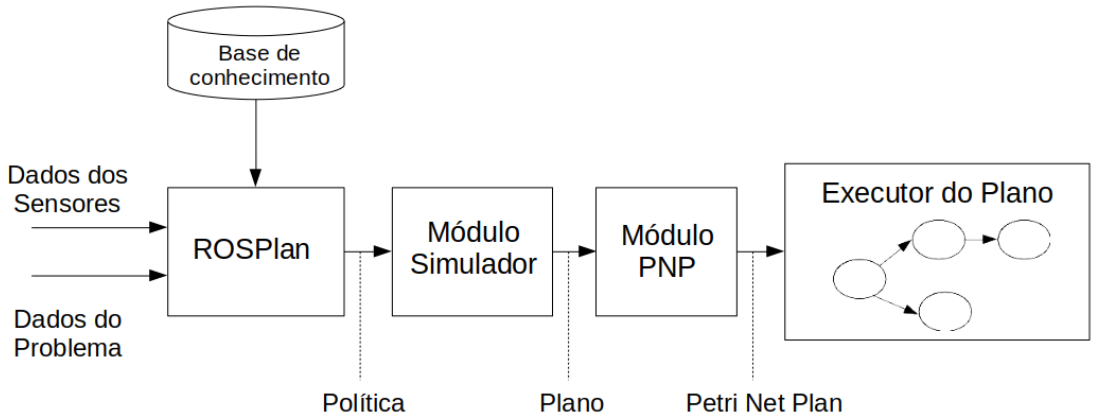

Figura 9 – Funcionamento macro da arquitetura proposta e seus módulos individuais

.

Toda a solução será construída dentro do ambiente R.O.S., desde a captura dos dados dos sensores até a ação enviada para cada outro módulo ou atuador. A seguir será discutido individualmente a proposta de modificação e os acréscimos de cada módulo do sistema.

5.1 MÓDULO ROSPLAN

Este módulo do sistema será responsável por selecionar a ação que maximize o ganho do agente no problema a cada etapa de iteração. O módulo, já disponível para alguns planejadores clássicos, terá que ter algumas estruturas internas modificadas para funcionar com o planejador probabilístico proposto e a imagem 10 ilustra as principais modificações, onde foi realizada a alteração no módulo que trabalha com a interação com a base de conhecimento e os sensores e também a alteração do despacho das ações, que agora são traduzidas em uma rede de Petri.

Na figura 10 a base de conhecimento, composta pelo analisador dos sensores, o módulo de raciocínio e a ontologia, foi modificada para lidar com novos atributos gravados na base de conhecimento. Já o sistema de planejamento, composto pelo gerador de problema, o planejador e o despacho das ações, foram modificados com a inclusão da etapa de tradução das ações despachadas em uma rede de Petri.

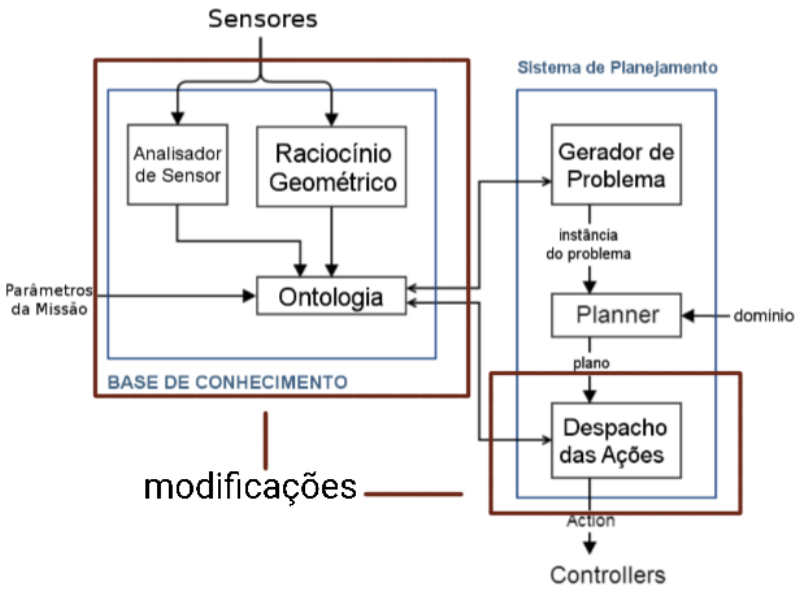

Figura 10 – Ilustração dos módulos modificados no *framework* do ROSPlan, sendo eles: A base de conhecimento e o despacho das ações

.

O planejador Prost-DD foi construído para funcionar com a linguagem de planejamento RDDL, como já descrito anteriormente, e a primeira modificação interna é na base de conhecimento, adequando a interpretação da arquitetura do ROSPlan para o novo modelo de linguagem de planejamento. Em seguida, como o planejador proposto funciona fornecendo uma política, necessitamos modificar o despacho das ações, uma vez que vamos utilizar uma tradução (módulo seguinte) para gerar todas as ações necessárias para o agente chegar ao objetivo.

De maneira geral, este módulo é a inteligência da arquitetura, recebendo a situação problema, os dados dos sensores e os dados da base de conhecimento (conhecimento do mundo) para fornecer uma política de seleção de ação.

5.2 MÓDULO SIMULADOR

Este módulo é responsável por realizar a composição do plano com base na política resultante do módulo ROSPlan. Essa política irá nos dizer a melhor ação para aquelas condições do estado atual, contudo precisaremos simular os próximos estados possíveis com base nas consequências que cada ação terá no ambiente e assim podermos compor o plano completo. Esse algoritmo de tradução é semelhante ao usado em (IOCCHI et al., 2016) para traduzir uma política em uma máquina de estados e funciona empilhando e explorando os estados iterativamente.

Os estados s são explorados consultando a política e assim tendo como retorno uma ação a. Quando a ação não é encontrada, o estado não é tratado pela política e possivelmente é um estado morto. Posteriormente é realizada a simulação de cada consequência e dentre o conjunto de possibilidades da ação a que leva a outro estado s'. Em seguida é analisado

se s' é o objetivo, se for o estado é tratado com um estado objetivo e verificamos se existem outros objetivos. Caso este estado s' não tenha sido explorado anteriormente, ele é adicionado aos estados visitados e assim sucessivamente. O algoritmo termina quando todos os estados alcançáveis são explorados, até que todos os objetivos tenham sido encontrados.

5.3 MÓDULO PNP E EXECUTOR DO PLANO

Este módulo é responsável por gerar a Rede de Petri que iremos utilizar para controlar no próximo estágio da arquitetura. O algoritmo proposto em (ZIPARO et al., 2011) traduz um plano (provindo da ferramenta ROSPlan por exemplo) em uma Rede de Petri, mas como iremos fornecer uma sequência de estados e suas transições, o algoritmo será modificado para receber esses dados e montar a Rede de Petri que será controlada pelo próprio pacote da ferramenta em conjunto com as *actions* do *framework* R.O.S.

Com a tradução em uma Rede de Pétri podemos utilizar transições baseadas em mapeamento de um acontecimento, utilizando então uma lógica condicional para tratar incertezas no ambiente sem a necessidade do replanejamento. Com isso o executor irá controlar o que está sendo executado, ação a ação até os objetivos, enquanto constantemente analisa os dados dos sensores.

6 DESENVOLVIMENTO

6.1 TRATAMENTO DO TEXTO DO GERADOR DE AÇÕES DA *ROBOCUP*

Como estamos utilizando as tarefas geradas pelo gerador de tarefas da RoboCup, que produz as tarefas em forma de texto, necessitamos transformar as requisições do gerador de tarefas em fluentes de objetivos em RDDL, foi desenvolvido um algoritmo baseado em análise semântica, que interpreta o texto, cria uma árvore de relação entre as palavras e extrai desta os objetos diretos e passivos vinculados a cada verbo imperativo da frase.

Quando o texto é gerado, é usada a biblioteca *SpaCy* da linguagem de programação *python*, cujo objetivo é atribuir a partir de um modelo de linguagem pré treinado cada significado sintático das palavras e assim a relação entre elas, como é ilustrado na figura 11. A partir desta construção a árvore sintática então passa pelo algoritmo desenvolvido, que interpreta os predicados, objetos e adjuntos adverbiais vinculados a cada verbo imperativo e atribui a cada ação os objetivos.

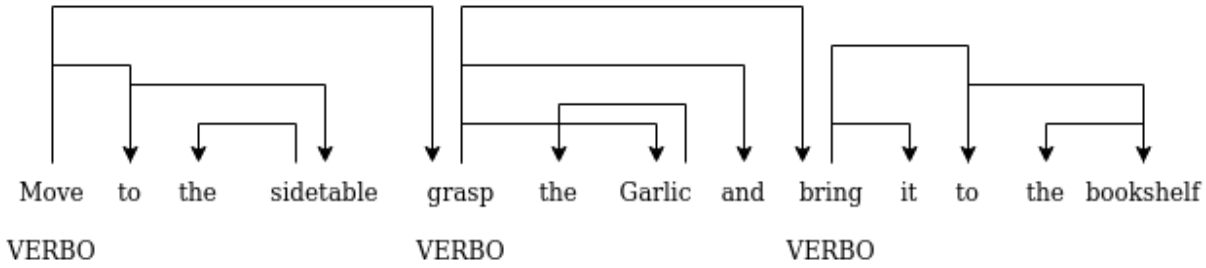

Figura 11 – Relação semântica entre as palavras do gerador da *Robocup@Home* utilizando a biblioteca SpaCy do Python

.

No exemplo acima, o resultado extraído da análise sintt́ica pode ser visto na figura 12.

O algoritmo 1 ilustra a composição do exemplo anterior. A parir da análise sintática, cada termo da oração é verificado quanto ao seu tipo e quando ocorrer um verbo, usamos um "DE PARA"de verbos para fluentes já mapeados e ainda resgatamos da árvore o objeto que está sofrendo a ação deste verbo. O retorno do algoritmo então é uma lista contendo os fluentes

```
(mestrado) leonardo@leonardo-notebook:~/Documents/Mestrado/gpsr_command_generator/main$
 python3 text_to_fluent.py
                ##########################################
                ############ FLUENT GENERATOR ##############
                ##########################################

 Phrase: Move to the sidetable, grasp the Garlic sauce and bring it to the bookshelf.

        Extracted relation from text: Move -> sidetable
        Extracted relation from text: grasp -> sauce
        Extracted relation from text: bring -> sauce -> bookshelf

        FLUENTS:
                ROBOT_GOAL(sidetable)
                GRASP_OBJECT(sauce)
                OBJECT_GOAL(sauce, bookshelf)
```

Figura 12 – Resultado do algoritmo que extrai os fluentes da frase de comando para o robô

.

objetivos ("Vá para"por exemplo) e seus argumentos ("cozinha"por exemplo).

Algoritmo 1 – Algoritmo para extração dos objetivos do agente a partir de uma frase.

Entrada: α Árvore sintática, DF De Para de Fluentes
Saída: λ Fluentes
Função `obtemFluentes`(α, DF)**:**
  $\lambda*$**;** Lista de objetivos
  Para cada $palavra$ *em* α **faça**
    Se $tipo(palavra)$ *é* $VERBO$ **Então**
      $\lambda* \longleftarrow DF(palavra), ObjetoVerbal(palavra);$
    Senão
  Fim
  retorna $\lambda*;$
End Function

6.2 MODELO DE MUNDO EM RDDL

Para modelar as ações, efeitos e objetos que compõem o mundo em RDDL, necessitamos de duas estruturas básicas: A Instância e o Domínio. A instância é onde descreve-se as variáveis de domínio "não fluentes", as condições iniciais do mundo e as configurações do MDP como horizonte limite, que é a quantidade de eventos limite para a resolução do MDP, fator de desconto de transição de estado e a quantidade de ações permitidas concorrentes para a etapa de execução.

Já o domínio é usado para descrever as regras lógicas que regem o modelo de mundo e como os fluentes são atualizados a cada iteração dos eventos. é no domínio também que

descrevemos a função de recompensa que o agente terá a medida que os eventos acontecem no horizonte. A seguir serão abordadas as construções de cada uma dessas estruturas para o problema em ambiente doméstico.

6.2.1 Arquivo da Instância do Problema

A princípio, na instância do problema, deve-se descrever os tipos de objetos que são manipulados pelo domínio, os não fluentes e as configurações do MDP. A seguir, para o problema doméstico, foram criados tipos de objetos simples para descrevermos a maioria dos objetos físicos que o agente poderá interagir:

a) waypoint: tipo de objeto que descreve uma localização;
b) obj: tipo de objeto que descreve um objeto físico no ambiente;
c) robot: tipo de objeto que descreve o agente robótico;
d) person: tipo de objeto que descreve pessoas.

Esse objetos podem ser manipulados pelo arquivo de domínio e são usados na construção das regras que definem todo o comportamento do mundo. À cada um desses tipos foram atribuídos objetos físicos do mundo, assim poderemos associá-los quando formos descrever as regras de interação. A seguir temos o exemplo da atribuição, em RDDL, de cada objeto físico a seu respectivo tipo:

```
1  objects {
2      waypoint: {
3          garagem, quarto1, quarto2, banheiro1, banheiro2,
4          sala, cozinha, mesaCozinha, lixoCozinha
5          };
6      obj: {
7          biscoito, soda, bolsa, lixo, sapato, salgadinho,
8          celular, travesseiro, notebook, mochila, copo,
9          xicara, livro, carteira
10         };
11     robot: {hera};
12     person: {
13         operador, morador1, morador2, vizinho1, vizinho2
14         };
15     }
```

Como o buscamos resolver um problema onde existe um objetivo a ser alcançado pelo agente, foi necessário o uso de não fluentes que descrevessem cada objetivo particular, que são extraídos da frase gerada pela ferramenta da *RoboCup@Home*. A seguir temos alguns exem-

plos dessas constantes:

```
1  non−fluents {
2          ROBOT_GOAL(mesaCozinha);
3          DELIVERY_GOAL(obj, person);
4          OBJECT_GOAL(celular, lixoCozinha);
5          FIND_SOMEBODY_GOAL = true;
6          INTRO_TO_SOMEBODY_GOAL = true;
7      };
```

Esses fluentes descrevem as condições que precisam ser verdadeiras para cada execução e resolução do problema, por exemplo, se o objetivo da tarefa é jogar a latinha de refrigerante no lixo, o algoritmo precisará tornar verdade o fluente *OBJECT_GOAL(lata, lixoCozinha)*,ou seja, precisa existir um objeto do tipo *obj* lata e ele deve estar no objeto do tipo *waypoint* lixoCozinha, e assim sucessivamente para cada sub objetivo que vamos resolver.

Na instância descrevemos também as condições iniciais do nosso modelo de mundo. Aqui podemos usar todos os tipos de objetos criados anteriormente e alimentamos uma condição, ou estado, que o agente poderá encontrar o mundo quando o sistema iniciar. A seguir podemos ver um exemplo da construção:

```
1      init −state {
2          robot_at(hera, sala);
3          somebody_at(operador, quarto);
4          somebody_at(vizinho, garagem);
5          object_at(celular, mesaCozinha);
6          object_at(travesseiro, quarto);
7          saw_somebody_at(garagem);
8          undocked(hera);
9          localised(hera);
10     };
```

No trecho acima novos predicados são vistos, como: *robot_at*, *somebody_at* e *object_at* por exemplo. Esses predicados foram criados (e serão abordados com maiores detalhes na descrição do domínio) para descrevermos a interação de cada um dos objetos com o mundo, assim, quando queremos dizer que inicialmente existe um celular na mesa da cozinha, podemos simplesmente criar o predicado *object_at(celular, mesaCozinha)*. Aqui também foi criado um predicado que ajuda o agente a procurar pessoas no ambiente, o *saw_somebody_at(*waypoint)*,

que ajuda a descrever para o modelo que foi visto alguma pessoa em determinado ambiente (objeto do tipo *waypoint*), mesmo sem ser reconhecida pelo agente. Lembrando que para o modelo de mundo fechado, ou seja, onde não acrescentamos novos elementos durante a execução da resolução do problema de forma dinâmica, necessitamos de recursos para que o agente consiga lidar com situações que não foram previstas, como é o caso de pessoas que não foram escritas antes da execução do problema.

Para qualquer problema as condições iniciais fazem-se necessárias e podem ser descritas depois do que já foi configurado anteriormente. Para nosso problema o estado inicial do mundo foi descrito como o trecho de código a seguir:

```
1   init −state {
2       robot_at(hera, sala);
3       somebody_at(operador, quarto);
4       somebody_at(vizinho, garagem);
5       object_at(celular, mesaCozinha);
6       object_at(travesseiro, quarto);
7       saw_somebody_at(garagem);
8   };
```

Aqui, analisando o trecho de código anterior, vimos que cada construção que descrevemos no estado inicial serão verdadeiras e as demais que não são descritas serão **Falsas**. Foram criadas condições para cada estado dos objetos que compõem o mundo e serão descritas a seguir no arquivo de domínio, mas um exemplo com o trecho anterior são os termos *robot_at, object_at e somebody_at*, que descrevem a posição do robô, de objetos e de pessoas no mundo.

Por fim, foram definidas três variáveis que complementam a instância do problema. A primeira, *max-nondef-actions*, configura o número de ações concorrentes que podem ser executadas a cada iteração de execução. A segunda, *horizon*, define o horizonte de eventos finitos que o agente terá para atingir a maior recompensa possível para aquele problema e por fim, a variável *discount* representa o desconto de cada iteração que o agente faz, influenciando diretamente na recompensa final. Essas variáveis foram configuradas como 1, 8 e 3 respectivamente.

6.2.2 Arquivo do Domínio

O arquivo de domínio, que pode ser acessado na íntegra em: `https://github.com/leocneves/action_planning/blob/main/domains/hera_domain.rddl`, foi quebrado em quatro grandes partes, as *pvariables* que representam as descrições dos não fluentes e dos fluentes de estado, o *cpfs*, onde descrevemos as iterações dos fluentes de estado a cada iteração do agente com o mundo, a *reward function*, que representa a recompensa do agente

a cada iteração e as *action-preconditions*, onde descrevemos cada condição prévia dos não fluentes para que uma ação (ou fluente de estado) seja tomada como verdadeira (ou executada propriamente dita).

6.2.2.1 As pvariables

Dentro das *pvariables*, ou variáveis que serão usadas no nosso problema pelo algoritmo de planejamento, foram criadas basicamente 3 tipos: *non-fluents*, *state-fluents* e *action-fluents*.

Dois exemplos das *non-fluents* criadas podem ser vistas no trecho de código a seguir:

```
1    ...
2        ROBOT_GOAL(waypoint): { non−fluent, bool, default = false };
3        OBJECT_GOAL(obj, waypoint): { non−fluent, bool, default = false };
4    ...
```

No exemplo acima, a variável *ROBOT_GOAL* representa um lugar no mapa que será o objetivo do agente, sendo assim, o planejador irá se esforçar para guiar o agente ao cumprimento desta condição se houver recompensa para a mesma (a recompensa será discutida posteriormente). De modo semelhante, a variável *OBJECT_GOAL* foi criada para informar ao planejador que esta condição precisa ser verdadeira quando o problema é colocar um objeto em determinada posição do mapa. Com isso podemos ver que ambas as definições são usadas, de maneira estratégica, para informar ao planejador as situações que precisam ser verdadeiras dependendo do problema que estamos lidando.

De modo semelhante, com dois exemplos, podemos ver como foram criadas as variáveis de estado fluentes, ou *state-fluents*.

```
1    ...
2        carrying_object (robot, obj): { state −fluent, bool, default = false };
3        saw_somebody_at(waypoint): { state −fluent, bool, default = false };
4    ...
```

Ambas as descrições no trecho descrito foram criadas para descrever condições que se alteram com base em ações que o agente executa ou em respostas que os sensores geram. A primeira variável do exemplo, a *carrying_object*, foi criada para informar ao planejador que alguma ação provocou o carregamento de algum objeto pelo agente, assim ações como "soltar objeto", que serão descritas a seguir, podem ser executadas. Diferente da primeira variável de estado, que foi criada e é atualizada por alguma ação (pegar um objeto neste caso), a segunda

variável descreve uma variável que foi criada para informar onde foram vistas pessoas no ambiente e sua atualização é realizada por sensores, visto que a câmera captura a presença de uma pessoa e este fluente é atualizado no para verdadeiro onde a pessoa se encontra, por exemplo: *saw_somebody_at(GARAGEM)*, onde a condição de o agente ter visto alguém na garagem se torna verdadeira para o modelo.

Por fim, as *action-fluents* podem ser vistas em um trecho a seguir. Elas descrevem as ações que o agente poderá escolher a cada iteração do problema respeitando o horizonte de eventos definido anteriormente e o número máximo de ações que podem ser executadas de maneira concorrente por iteração.

```
1   ...
2       // Habilidades da Hera
3       goto_waypoint(robot, waypoint, waypoint): { action-fluent, bool,
4                                                   default = false };
5       grasp_object(robot, obj, waypoint): { action-fluent, bool, default = false
            };
6       place_object(robot, obj, waypoint): { action-fluent, bool, default = false
            };
7       // Pessoas
8       aproximateToSomebody(robot): { action-fluent, bool, default = false };
9       introMyself(robot): { action-fluent, bool, default = false };
10  ...
```

De modo geral, o agente a cada iteração pode escolher a *action-fluent* necessária que irá lhe dar a maior recompensa possível e é muito importante que todas as ações do agente (ou habilidades) estejam bem definidas, para que o mesmo consiga atingir estados objetivos de maneira ótima.

Na situação descrita no trecho anterior são exemplificados quatro habilidades que a robô *HERA* pode desempenhar no ambiente. A primeira, *goto_waypoint*, é uma ação para que ela se locomova entre dois ambientes, sendo o seu atual e um objetivo. A segunda e a terceira, *grasp_object* e *place_object*, descrevem respectivamente as ações de pegar e soltar algum objeto e são úteis para o agente manipular ou carregar objetos pelo ambiente. As últimas duas ações são relacionadas com pessoas, a *aproximateToSomebody* e a *introMyself*, e tem por objetivo interagir com alguém nas proximidades do agente, sendo a primeira para simplesmente se aproximar de alguém e a segunda para realizar uma apresentação do próprio agente à pessoa objeto da ação.

6.2.2.2 Descrição das CPFs *e a* Reward Function

Para criar os fluxos de atualização de cada fluentes de estado e fluente das ações, necessitamos descrever quando cada fluente mudará seu estado e quais condições anteriores necessitam ser verdadeiras. Foi utilizado o CPF, ou *conditional probabilistic function*, como tipo de transição de fluxo da RDDL, pelo fato de que podemos ter distribuições em variáveis específicas e teremos atualizações e fluentes como regras condicionais.

Como descrito anteriormente, para cada fluente de estado ou de ação devemos descrever sua atualização para o próximo estado, assim com três exemplos a seguir poderemos entender como isso foi feito no detalhe.

```
1  cpfs {
2      robot_at '(?r, ?w) =
3      if (exists_{?w1: waypoint} (goto_waypoint(?r, ?w1, ?w))) then true
4      else if (exists_{?w1: waypoint} (goto_waypoint(?r, ?w, ?w1))) then false
5      else robot_at(?r, ?w);
6
7      object_at '(?o, ?w) =
8      if (exists_{?r: robot} (place_object(?r, ?o, ?w))) then true
9      else if (exists_{?r: robot} (grasp_object(?r, ?o, ?w))) then false
10     else object_at(?o, ?w);
11
12     carrying_object '(?r, ?o) =
13     if (exists_{?w: waypoint} [grasp_object '(?r, ?o, ?w)]) then true
14     else if (exists_{?w: waypoint} [place_object '(?r, ?o, ?w)]) then false
15     else carrying_object(?r, ?o);
16     ...
```

No trecho acima vemos três descrições de atualizações de fluentes que foram feitas para o problema: A atualização da posição do robô (*robot_at*), a atualização da localização de um objeto no mundo (*object_at*) e a atualização do fluente que diz se o agente está carregando algum objeto (*carrying_object*). Para cada uma dessas descrições formais necessitamos descrever exatamente o que precisa estar acontecendo no estado atual para que o fluente possa alterar seu valor de estado.

Para a primeira descrição de atualização, o RDDL cria o fluente *robot_at* para cada possível lugar no mundo, mas só será verdadeiro o que de fato o robô estará e assim, quando o robô mudar de posição, esse mesmo terá que ser falso. Essas atualização são descritas nas linhas

3, 4 e 5 do trecho acima. A linha 3 verifica se no estado atual existe um fluente *goto_waypoint* para o *waypoint* em questão e se sim, ou seja, se o robô teve um comando de ir a um ponto no mapa, essa regra torna verdadeiro o termo *robot_at* para aquele *waypoint*. De modo semelhante, a linha 4 mostra o contrário, quando o agente recebe o comando de *goto_waypoint* e sai do lugar atual, a regra desta linha atualiza o *robot_at* do ponto atual para falso e por fim, na linha 5 é representado que se não houver atualizações pela ação *goto_waypoint*, os fluentes *robot_at* ficam como estão.

De modo semelhante ao descrito com a atualização da posição do robô para cada possível ambiente do mundo, as linhas 7, 8, 9 e 10 descrevem a atualização de cada objeto no mapa, mas desta vez os fluentes de ação *grasp_object* e *place_object* que fazem a atualização.

As linhas 12, 13, 14 e 15 descrevem a atualização do fluente responsável pelo estado de carregamento do manipulador, que é usado pelas ações de pegar um objeto e soltar. De modo semelhante ao *object_at*, o *carrying_object* é atualizado pelos fluentes *grasp_object* e *place_object*, sendo que *carrying_object* se torna verdadeiro no estado futuro quando *grasp_object* é verdadeiro no estado anterior e se torna falso no estado futuro quando *place_object* é verdadeiro no estado anterior.

Vale ressaltar que no RDDL o fluente de estado futuro é descrito como o próprio nome do fluente somado a uma aspas simples, assim, se no estado atual descrevemos as posições do robô nos pontos do mundo como: *robot_at*, a descrição das posições para o estado futuro descrevemos como: *robot_at'*.

A função de recompensa é responsável por atribuir um ganho ao agente a cada ação tomada em cada estado. Como nosso problema no ambiente doméstico envolvem objetivos bem definidos, a função recompensa foi criada para que quando esses objetivos forem atingidos o agente possa ganhar recompensa máxima. A seguir podemos ver a função de recompensa que o algoritmo usa:

```
1 reward =
2     if (goal_obj ^ goal_find_somebody) then 3000
3     else if (~goal_obj) then (sum_{?r: robot, ?o: obj, ?w: waypoint, ?w2: waypoint}
          [( place_object (?r, ?o, ?w) ^ OBJECT_GOAL(?o, ?w))] * 100)
4     else if (~goal_find_somebody) then (sum_{?r: robot}
          [(FIND_SOMEBODY_GOAL ^ aproximateToSomebody(?r))] * 500)
5     else (-200);
```

Com o propósito do agente atingir os objetivos necessários, não fluentes foram criados de maneira estratégica para sinalizar o que o algoritmo tem que buscar na resolução do problema. No trecho acima, os não fluentes *FIND_SOMEBODY_GOAL* e *OBJECT_GOAL* são responsáveis por sinalizar quais são os objetivos principais e assim o agente ter a recompensa

máxima ao tornar estes verdadeiros. A estratégia desenvolvida para as tarefas de propósito geral da *Robocup@Home* foi a atualização da função de recompensa com os não fluentes atualizados de acordo com a etapa de extração dos objetivos da frase gerada pelo gerador oficial e que pode ser vista a diante.

6.2.2.3 Descrição das action-preconditions

As *action-preconditions*, ou pré condições das ações, são estruturas lógicas que descrevem as condições de alguns fluentes que precisam ser atendidas para que os fluentes de ações sejam verdadeiros e consequentemente as ações do agente sejam executadas. A descrição das condições foram feitas em linguagem lógica e a seguir podemos ver alguns exemplos para analisar no detalhe:

```
1   action-preconditions {
2       // O agente so pode se aproximar se ele viu alguem no comodo
3       forall_{?r: robot} [aproximateToSomebody(?r) => (exists_{?w: waypoint}
            [(robot_at(?r, ?w) ^ saw_somebody_at(?w))])];
4       // O Agente precisa estar no lugar para pegar/colocar um objeto
5       forall_{?r: robot, ?w: waypoint, ?o: obj, ?o2: obj} [grasp_object(?r, ?o,
            ?w) => (robot_at(?r, ?w) ^ object_at(?o, ?w) ^ ~carrying_object(?r,
            ?o2))];
6       forall_{?r: robot, ?w: waypoint, ?o: obj} [place_object(?r, ?o, ?w) =>
            (robot_at(?r, ?w) ^ carrying_object(?r, ?o))];
7       // O agente nao pode estar em dois lugares ao mesmo tempo
8       forall_{?r: robot, ?w1: waypoint, ?w2: waypoint} [?w1 == ?w2 |
            (robot_at(?r, ?w1) => ~robot_at(?r, ?w2))];
9   ...
```

Acima podemos verificar algumas regras que foram desenvolvidas e precisam ser atendidas para tornar o modelo de mundo coeso. Na linha 3 podemos ler o trexho como: "Para todo robô, deve existir um ponto no mapa em que este robô esteja e que ele tenha visto alguém la para que o fluente *proximateToSomebody(?r)* seja verdadeiro (ou executada)". Aqui, estamos falando para a lógica do modelo de mundo que para o robô se aproximar de alguém em algum lugar, ele precisa estar neste lugar e ter alguém la que ele já tenha visto.

Assim como foi feita a regra descrita acima da linha 3, as linhas 5 e 6 descrevem as regras para que o agente pegue/solte algum objeto e podem-se ser lidas como: "Para todo robô, ponto no mapa e objeto no mapa, os fluentes *grasp_object* e *place_object* só serão verdadeiros, se e somente se, o robô estiver no ponto do mapa em que o objeto está e não estiver/estiver

carregando algum objeto. Por fim, no último exemplo do trecho de código acima, a linha 8 descreve a regra em que o agente não pode estar em dois lugares ao mesmo tempo e pode-se ser lida como: Para todo robô, ponto 1 no mapa e ponto 2 no mapa, o fluente *robot_at* para o ponto 1 no mapa é verdadeiro, se e somente se, o fluente *robot_at* é falso para o ponto 2 no mapa.

6.3 ESTRUTURA *ROSPLAN* E *PETRI NETS*

Nesta seção será descrito o desenvolvimento e as alterações na parte principal do sistema, o planejamento das ações. A estrutura basicamente funciona em três pilares principais, a simulação das ações para construção do plano a partir da função de recompensa, a conversão do plano linear para um plano que contem regras de execução para evitar falhas e replanejamentos futuros e a chamada que executa o planejamento novamente, caso algo inesperado ocorra, e agrega a nova ramificação á estrutura anterior, criando assim um plano mais robusto para execuções futuras. Os detalhes de cada etapa serão tratados a seguir.

6.3.1 Planejamento das Ações

Na etapa de planejamento das ações, foi utilizado o algoritmo de planejamento probabilístico PROST-DD. Recentemente o PROST já havia sido introduzido ao ROSPlan, trazendo a possibilidade da construção e modelagem de problemas na linguagem RDDL, porém para o trabalho atual o algoritmo foi alterado no *launch* principal do ROSPlan, devido às características da função de seleção das ações do PROST-DD.

Para a estrutura proposta, que utilizamos a tradução do plano para uma rede de Petri, necessitamos configurar o ROSPlan para o tipo de *dispatcher* simples, que executa as ações uma a uma e gera um plano ao final da requisição. Nessa configuração, aproveitamos a interface com o RDDL no ROSPlan e criamos a chamada do planejador PROST-DD, que pode ser encontrado em seu repositório aberto no github (`https://github.com/leocneves/action_planning`), em um *launch* no sistema para robôs ROS. Nesse mesmo *launch* principal foram configuradas as chamadas para o domínio e a instancia em RDDL, como detalhado anteriormente e o código completo se encontra no repositório do projeto.

A simulação das ações para geração do plano é feita utilizando a mesma ferramenta usada na competição de planejamento IPPC. A ferramenta, a RDDLSim, já havia sido integrada ao ROSPlan e usamos essa interface para gerar a sequência de ações que pode ser usada pelas fases seguintes da arquitetura proposta. A imagem a seguir ilustra a integração de cada um dos blocos mencionados e a saída desta etapa para a seguinte, onde a instância e o domínio (em RDDL) são utilizados pelo ROSPlan para montar a base de conhecimento e em seguida realiza a execução do RDDLSim, para simular o problema, com o PROST configurado para gerar o plano (sequência de ações) que será utilizado pela próxima etapa.

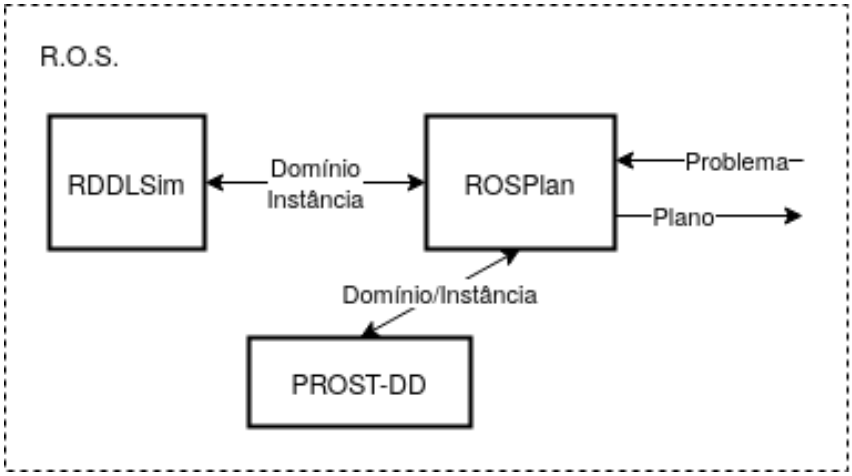

Figura 13 – Comunicação dos módulos de entrada da estrutura proposta.

.

6.3.2 Redes de Petri e Regras de Execução

A etapa seguinte da arquitetura, a partir de um plano da etapa anterior já descrita, constrói uma rede de Petri a partir de uma sequência de ações e adiciona efeitos concorrentes, habilitando fluxos concorrentes a partir das regras de execução. Para detalhar esta construção vamos utilizar um comando simples ao robô: "Go to the garage and wait".

A partir desta frase, as ações necessárias para atingir o objetivo são: "goto_waypoint(garage)"e "wait". A construção da rede de Petri é feita adicionando as transições como início e conclusão das ações e os lugares representam a execução em andamento da ação. A imagem 14 ilustra como a tradução do exemplo anterior poderia ser feita deste modo.

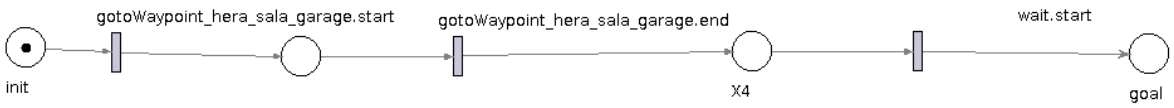

Figura 14 – Tradução do exemplo "Go to the garage and wait"para uma Petri Nets.

.

Com isso a tradução é feita, mas para colaborar a corrigir o problema dos estados mortos, vamos adicionar as regras de execução e tornar o plano mais robusto adicionando regras que possibilitam realizar ações concorrentes com o plano atual e minimizando assim os impactos que interações paralelas com o robô ou objetos não esperados no caminho teriam. Continuando com o exemplo anterior, vamos considerar a seguinte regra de execução:

```
1    *if* personhere *during* goto_waypoint
2    *do* say_hello ;  waitfor_not_personhere ;  restart_action
```

A regra descrita fala que, se houver alguma pessoa presente, condição *personhere*, durante qualquer ação de mover-se a algum ponto no mapa (*goto_waypoint*), a transição concorrente é acionada e o robô executa as ações sequenciais de falar oi (*say_hello*), esperar a pessoa sair do caminho (*waitfor_no_personhere*) e então retorna para a ação que estava sendo executada (*restart_action*). A figura 15 ilustra a situação onde durante a execução de uma ação, uma transição de percepção pode ser acionada para executar uma segunda ação. Entre a transição de percepção e a transição de começar a ação 2, temos os lugares de transição, que não tem efeito nenhum nas execuções do agente e serve apenas como instrumento de ligação entre as transições de percepção e de começo das ações.

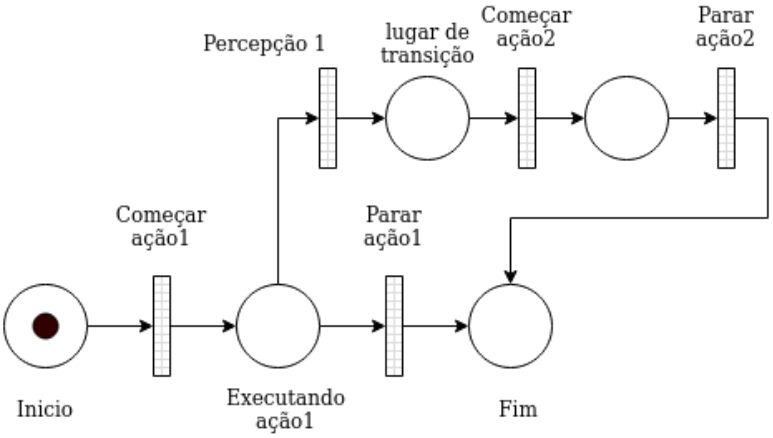

Figura 15 – Trecho de exemplo de um dos comandos em Petri Nets que contém regras de execução, com mais de uma transição podendo ser acionada de um lugar.

.

Assim, as regras de execução tornam a estrutura mais robusta, umas vez que elas podem criar um monitoramento entre duas ações. Exemplo disso é o deslocamento do robô entre dois ambientes, onde para o planejamento, quando o robô chega ao segundo ponto uma nova ação será executada criando assim um tempo "cego"entre as execuções. Com a criação de uma transição que pode ser acionada durante esse período, as ocorrências do mundo que podem resultar em um estado morto podem ser minimizadas, como por exemplo um grupo de pessoas fechar o caminho do robô e ele nunca finalizar a ação planejada.

6.3.3 Replanejamento e Agrupamento de Plano

Para uma maior flexibilidade do agente e para que possamos reagir caso ocorra ainda casos que nem mesmo o plano inicial ou as regras de execução possam lidar, aqui será apresentada a estratégia desenvolvida para que o agente possa chamar o planejador novamente e tentar atingir o objetivo lhe dado.

Até agora foi apresentado como podemos resolver problemas de estados mortos planejando as ações e aplicando regras que possibilitam que o agente possa lidar com condições adversas no meio destas execuções, quando o conhecimento do mundo fica limitado, mas mesmo

assim pode ser que as premissas iniciais que o planejador adotou com base no conhecimento do mundo naquele momento leve o agente a estados que não necessariamente os objetos ou pessoas estejam lá para que ele possa interagir, por exemplo. Aqui o replanejamento faz-se necessário e somente com a estrutura já mencionada não havia uma estratégia clara de como poderíamos fazer isso, dado que o robô pode já ter conseguido solucionar parte do problema inicial e assim no novo plano teríamos que lidar com esse tipo de problema, o de encorporação do que já foi feito.

A partir disso a estratégia adotada foi a de que a cada problema de replanejamento que possa existir, a rede de Petri já gerada não será apagada e sim, a partir dela será gerado um novo plano com as regras de execução já mencionadas anteriormente e assim um novo ramo será criado para aquela condição. Foi pensada essa estratégia para que no futuro, dadas as mesma condições locais que foi forçado o replanejamento, o agente possa utilizar a estrutura já gerada para continuar a execução das ações até o objetivo, economizando assim tempo de geração do plano para casos adversos. O pseudo código a seguir mostra como é feita essa expansão do plano já gerado mediante à necessidade de consulta ao planejador novamente.

Algoritmo 2 – Função para aumentar o plano já existente com uma nova consulta ao planejador

1 **Entrada:** λ Plano atual, s Estado atual, G Objetivos, ER Regras de Execução
2 **Saída:** $\lambda*$ Plano expandido
3 **Função** `expandePlano(`P`)`**:**
4 λ^1 **= chamadaPROST(**s, G**)** $\lambda*$**;**
5 **Para cada** $\text{ação} \in \lambda^1$ **faça**
6 **Se** $\text{ação} \text{está} \text{em} ER$ **Então**
7 $\lambda* \longleftarrow ER(\text{ação})$;
8 **Senão**
9 $\lambda* \longleftarrow \text{ação}$;
10 **Fim**
11 **Fim**
12 **retorna** $\lambda*$;
13 **End Function**

7 RESULTADOS

Para testar a arquitetura proposta, foram geradas 6 tarefas, duas para cada um dos grupos descritos a seguir, com o gerador da *Robocup@Home*, onde o ambiente usado para os testes está ilustrado na figura 16 que representa o ambiente do laboratório em que o projeto do robô HERA é desenvolvido. O conjunto de ambientes da casa, objetos e ações foi mantido, como proposto no repositório oficial do gerador de tarefas, e apenas foram alocados em cada ambiente do laboratório, que na figura 16 são representados pelo seu nome e um ponto de cor verde e vermelha no mapa.

Cada uma das tarefas listadas a seguir foi realizada 4 vezes, mediante à analise da média e desvio padrão dos dados coletados, em ambiente simulado e os indicadores descritos a seguir, foram calculados pela média simples destas execuções, pois as buscas no espaço de estados que o planejador realiza nem sempre são realizadas no mesmo tempo.

Grupo A) Aõs com deslocamento e manipulação de objetos:
- Move to the hallway table, grasp the Coke and put it in the bookshelf.
- Move to the kitchen counter, grasp the Apple juice and bring it to me.

Grupo B) Ações apenas com deslocamento:
- Move to the bench, move to the bookshelf, and move to the bar.
- Move to the bathroom, move to the bedroom, and move to the hallway table.

Grupo C) Ações com deslocamento e interação:
- Move to the garage, find a person, and introduce yourself.
- Move to the hallway table, find a person, and guide it to the exit.

Além dos experimentos propostos em condições em que não houve alterações no ambiente (as informações do modelo de mundo e condições iniciais do problema foram mantidas), foi realizada uma segunda rodada de experimentos com as mesmas tarefas, só que para cada grupo de tarefas as seguintes condições foram geradas para verificar a generalização da arquitetura para problemas adversos. Estas condições podem ser vistas a seguir:

a) Grupo A: o objeto foi retirado do local antes do agente tentar manipulá-lo.
b) Grupo B: enquanto o agente se desloca, uma pessoa entra na frente e permanece parada por um tempo.
c) Grupo C: a pessoa para quem o robô deve se introduzir foi mudada de posição.

E, por fim, em cada grupo de tarefas executado, foram analisados os seguintes parâmetros:

* Tempo de construção do plano (planejamento + simulação das ações);

* Tempo total da arquitetura (tempo decorrido desde o recebimento da tarefa até o início da primeira ação);
* Quantidade de vezes que houve replanejamento;

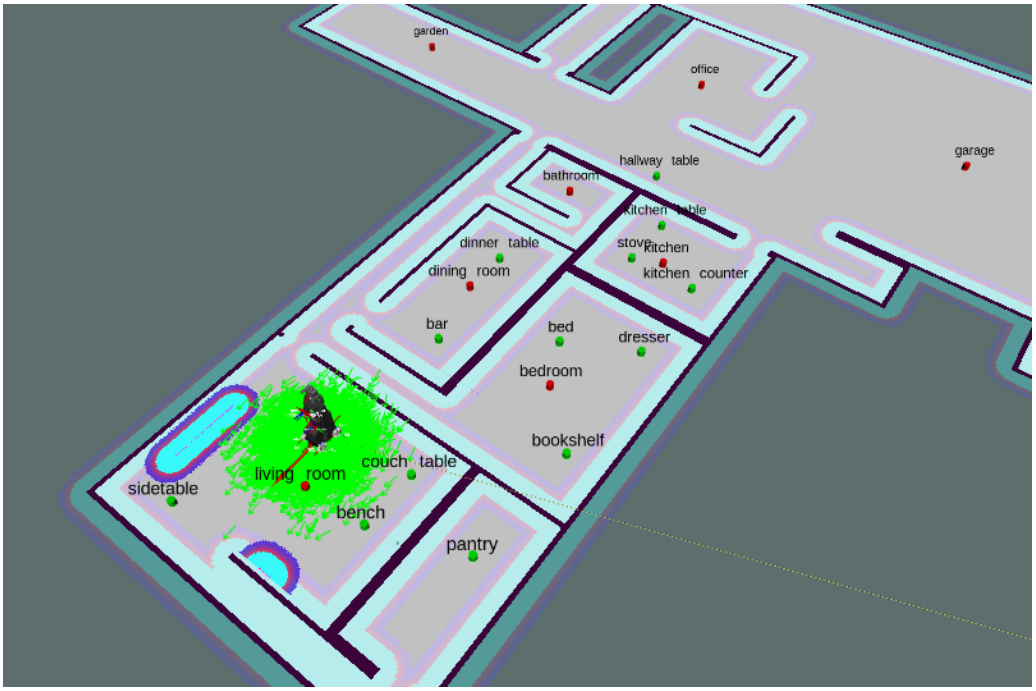

Figura 16 – Representação do mapa, percepção dos sensores e pontos de interesse para a simulação das tarefas.

.

A tabela 1 mostra os resultados obtidos, em média, em cada uma das rodadas realizadas, onde podemos ver para cada rodada e grupo de tarefas os tempos de planejamento, da arquitetura e a quantidade de vezes que foi necessário o replanejamento. A figura 17 ilustra um exemplo de rede gerada para a primeira tarefa, onde o número 1 representa o começo da execução da ação nas transições e o número 2 representa o fim de cada execução. Os estados de percepção de uma pessoa na frente funcionam, como mostra a imagem, de maneira concorrente durante a execução das ações de mover-se (MOVE) e assim são acionados estas ramificações (os lugares com nome de "Interação"na imagem) durante as execuções caso haja detecção. Todos os lugares na figura 17 representam execuções de ações e as transições representam começo e fim das execuções. As imagens originais das Redes de Petri geradas pela arquitetura para pelo menos uma rodada de experimentos, para as tarefas 1, 2 e 3, podem ser acessadas no endereço do projeto (`https://github.com/leocneves/action_planning`) com os nomes de "teste1.png", "teste2.png"e "teste3.png".

Podemos também verificar a dispersão dos tempos de execução em cada uma das tarefas dentre as rodadas de testes nas imagens 18 e 19. Nas imagens são mostrados os números da tabela 1 e além disso os desvios padrão atribuídos a cada grupo de tarefas

Tabela 1 – Dados dos testes realizados em ambiente simulado

	Grupo	*Tarefa*	*Tempo de construção do plano (s)*	*Tempo total da arquitetura (s)*	*Quantidade de vezes que houve replanejamento*
Rodada 1	A	1	12.46	27.46	0
		2	13.11	28.11	0
	B	1	13.23	28.23	0
		2	13.28	28.28	0
	C	1	14.39	29.39	0
		2	14.15	29.15	0
Rodada 2	A	1	15.67	33.67	1
		2	14.89	32.89	2
	B	1	15.32	33.32	1
		2	16.12	34.12	1
	C	1	18.46	37.46	2
		2	18.1	36.81	2

Fonte: Autor

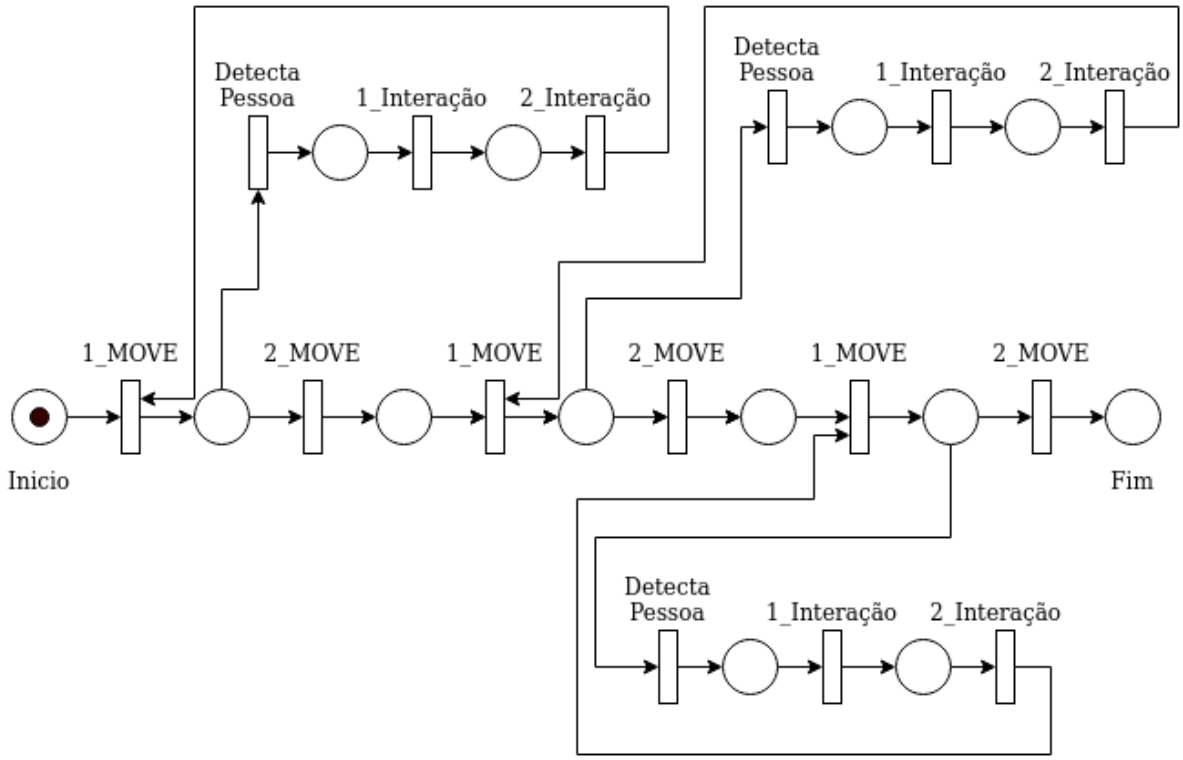

Figura 17 – Rede de Petri gerada a partir de experimentos do grupo A de tarefas.

.

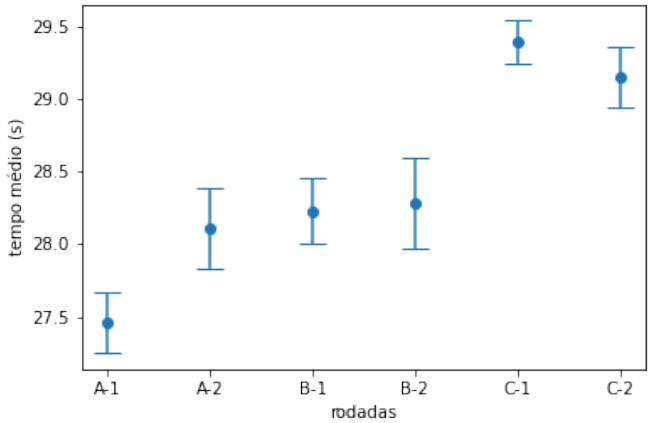

Figura 18 – Dispersão das médias de tempo obtidas nos experimentos e seus desvios padrão para a primeira rodada de testes.

.

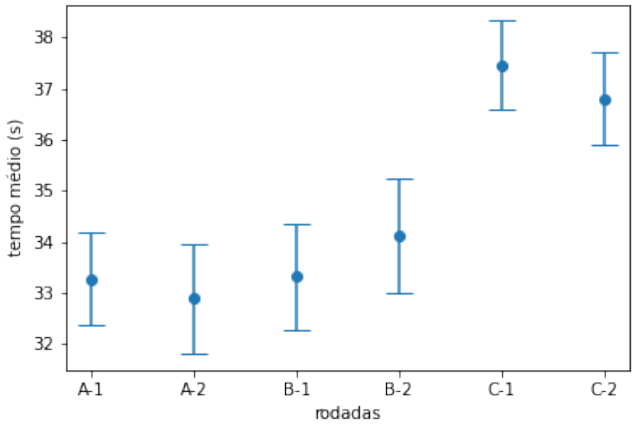

Figura 19 – Dispersão das médias de tempo obtidas nos experimentos e seus desvios padrão para a segunda rodada de testes.

.

8 CONCLUSÃO

A arquitetura proposta tem como função, a partir de uma tarefa geral dada ao agente, gerar um plano robusto e flexível para atingir este objetivo. Como visto na secção de desenvolvimento, a construção de um modelo que extrai os fluentes dos comandos por meio de árvores sintáticas, permite que logo de início possamos ter uma maior padronização de tudo que entra na estrutura, assim, reduzindo o esforço do planejador nas etapas seguintes. O modelo de mundo criado para o ambiente descrito é totalmente flexível e adaptável, permitindo que novos objetos, pessoas, lugares e suas distribuições de probabilidades sejam facilmente alterados ou adicionados, sendo que as principais ações que o agente tem até o momento, já foram adicionadas no modelo. Podemos ressaltar que a função de recompensa na estrutura também pode ser facilmente alterada, uma vez que pode-se ter a necessidade de recompensar o agente de maneira diferente, não só para conclusões de tarefas como o proposto neste trabalho, mas também para interações sociais ou outras atividades que o agente possa contribuir. Assim, com o modelo de mundo criado neste trabalho, podemos sanar o primeiro problema que antes existia: o problema anterior com o método de construção de tarefas orientadas a lógica condicional, ou diversas estruturas "se-então"no código do agente.

Dos problemas descritos na secção de revisão da literatura, notou-se que a arquitetura descrita neste trabalho mostrou-se eficiente em atingir ótimos locais, uma vez que em todos os experimentos um objetivo foi atingido, mostrou-se também eficiente em trazer atualizações no modelo de mundo de maneira concorrente com as execuções do plano e em expansões dos planos gerados caso haja replanejamentos. Esses aprimoramentos de trabalhos anteriores foram avaliados quanto ao seu funcionamento e tempo de execução, de modo que necessitamos de agentes que resolvam problemas em tempo hábil durante interações humano-robô.

Dos experimentos realizados e descritos em mais detalhes na seção de resultados, a tabela 1 mostra os resultados obtidos com a execução de cada uma das tarefas, na média, e podemos verificar os tempos que o planejador levou para resolver o problema, bem como o tempo total da arquitetura para iniciar a primeira ação. Em relação a variação dos tempos em cada uma das tarefas da primeira etapa, onde não foram alterados os objetos no sistema, vimos que todas as três tarefas apresentaram tempos semelhantes e o tempo total da arquitetura pode ser considerado não prejudicial em condições reais, uma vez que a tarefa com maior tempo total demorou 29.39 para ser planejada, menos que o necessário para inicio de um movimento na competição *RoboCup@Home*. Na tabela também pode-se notar também, que a etapa de construção da rede de petri, coluna de "Tempo total da arquitetura", a partir da sequência de ações do planejador é linear e não toma muito tempo, podendo-se notar que na média essa etapa demora 15 segundos para sua execução completa. O desvio padrão médio das execuções ficou em 0.232 segundos, mostrando pouca variação dos experimentos da média e é possível analisar cada um deles nas figuras 18 e 19.

A segunda rodada, onde objetos e pessoas foram mudados de posição mediante ao que o modelo de mundo conhecia em suas condições iniciais, apresentou replanejamentos para correção do plano inicial e mesmo assim, o tempo de execução total não foi muito distante das tarefas na primeira rodada e o desvio padrão médio ficou em 0.98, mostrando que com o aumento da complexidade em relação à rodada inicial a variação do planejador aumenta. As tarefas dos grupos A e C desta rodada que apresentaram 2 replanejamentos na média para uma mudança no sistema, tiveram um maior número de replanejamentos devido às distribuições de probabilidades das pessoas nos locais, onde para motivos de testes, foram adicionados percentuais iguais na frequência de aparição de uma pessoa em dois ou mais dos ambientes modelados e o planejador teve que selecionar um de maneira aleatória (No PROST a seleção é por Monte Carlo) para procurar no momento do planejamento. De modo geral, vimos que tanto a simulação das ações para gerar um plano inicial, o sistema que transforma o plano inicial em uma rede de petri e a etapa que cria variações para o plano, mesmo com a expansão e replanejamentos (executados na rodada 2 de experimentos), não tiveram grandes impactos no tempo de execução e na dificuldade de conclusão das tarefas, uma vez que todas as tarefas foram concluídas com sucesso em ambiente simulado.

Nos testes vimos que as regras de execução criadas, por exemplo o fluxo que cumprimenta e espera a pessoa sair do caminho para continuar a navegação, são fundamentais para contribuir com problemas que podem levar a estados mortos, uma vez que enquanto o robô realiza as ações, o mundo não para de ser alterado, seja pelas pessoas que estão ao redor ou por ações que não foram bem executadas (objeto que cai ao chão quando manipulado) e assim o simples fato de adicionarmos recursos que mapeiam essas condições de maneira concorrente à execução do plano principal acaba tornando a execução mais robusta e fluida. No caso de objetos por exemplo, a simples ação do objeto cair de volta na mesa na manipulação pode ser um problema para o planejador, uma vez que os horizontes são bem definidos antes das execuções e pela função de recompensa, a mesma ação seria executada, mas com uma ramificação que percebe essa condição de maneira concorrente com a execução da ação, o plano inicial não teria impacto e um fluxo de correção poderia ser executado para simplesmente alinhar o agente e executar a mesma ação, sem a necessidade de um novo replanejamento.

Em termos de percepções que foram notadas nas execuções e que podem ser melhoradas, foi visto que quando as tarefas tem a mesma ação para ser executada diversas vezes, a ordem da execução com base no comando dado ao agente não é respeitada. Por exemplo, para o comando "Mova-se para a cozinha e Mova-se para o bar", a hierarquia correta é o agente se movendo para a cozinha e da cozinha para o bar, contudo, como o planejador tenta maximizar a recompensa por cada execução, a ordem com que cada ação é executada não importa para o agente e assim a ação de mover-se para o bar pode, em algumas vezes, vir primeiro, dependendo da seleção inicial do planejador, pois ambas tem a mesma recompensa. O segundo ponto de atenção é a relação entre a quantidade de elementos no modelo de mundo e o tempo com que o planejador demora para resolver o problema. Foi feito um teste quando percebeu-se esse

problema para verificar as limitações do modelo proposto e para a tarefa de ir até a dispensa, pegar um refrigerante e colocar na mesa da cozinha foi variado a quantidade de objetos a mais no mundo. A figura 20 mostra a relação entre o aumento na quantidade de objetos no mundo e o aumento de tempo para produção do plano pelo planejador. De maneira geral, com o aumento dos estados possíveis para o planejador fazer as buscas e calcular as recompensas, fica mais difícil para otimizarmos o tempo total da arquitetura, porém podemos ter algumas estratégias para correção dessa desvantagem intrínseca do planejamento em trabalhos futuros, uma delas é o simples agrupando dos objetos em categorias, deixando o raciocínio do objeto para uma segunda camada da arquitetura, ou a alteração dos parâmetros do PROST, que no caso deste trabalho, foram usadas as configurações padrão dele e que são tidas como referência nas competições de planejamento.

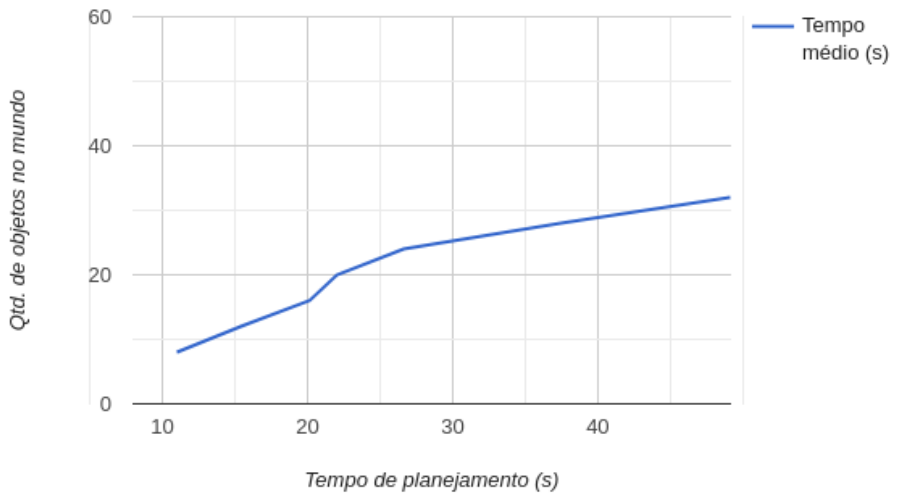

Figura 20 – Relação entre a quantidade de objetos no modelo de mundo e o tempo necessário para produção do plano para a tarefa: "Vá até a dispensa, pegue o refrigerante e coloque-o na mesa da cozinha"

.

Ao tratar sobre as melhorias que podemos corrigir nos pontos já citados anteriormente, em trabalhos futuros, um módulo que atualiza as frequências de aparição dos objetos no mundo pode ser implementado, uma vez que com isso pode-se reduzir o número de replanejamentos. A adição de uma camada de aprendizado de máquina por cima da arquitetura também poderia ajudar a tornar ainda mais robusta a estrutura e menos susceptível a replanejamentos, dado que cada plano para cada tarefa e cada condição inicial são guardados, uma estrutura poderia aprender qual plano se adapta a novos comandos com base nas experiências passadas e assim evitando até mesmo o uso do planejador em algumas situações.

Como o intuito da tradução do plano para uma Rede de Petri foi apenas de nos aproveitarmos da estrutura de lugares e transições, possibilitando a ocorrência de percepções do mundo

de maneira concorrente, não foram avaliadas as redes criadas quanto as suas propriedades, visto que não houve a necessidade para tal. Todas as redes criadas haviam apenas uma marca transitando pelos lugares, que representa a situação atual do sistema. Trabalhos futuros podem se aproveitar das redes criadas e análises mais aprofundadas podem ser feitas.

Por fim, vimos que a estrutura proposta teve um bom desempenho ao que se propôs, quanto ao tempo de execução e replanejamentos e pode ser aplicada à tarefas de propósito geral em ambiente doméstico ou de competições. O módulo de transformação de texto em fluentes do modelo de mundo foi um adicional deste trabalho e que pode ser alterado e melhorado no repositório aberto do projeto. Os testes em ambiente real, foram iniciados, mas devido às condições da pandemia do novo coronavírus e situações adversas do projeto, não foram finalizadas, mas não deixaram de validar o trabalho, uma vez que os resultados feitos com o planejador são os mesmo para ambos os ambientes e somente a execução das ações seriam alteradas.

REFERÊNCIAS

AERONAUTIQUES, Constructions et al. **PDDL| The Planning Domain Definition Language**. [S.l.], 1998.

ATRASH, Amin; KOENIG, Sven. Probabilistic Planning for Behavior-Based Robots. In: FLAIRS Conference. [S.l.: s.n.], 2001. p. 531–535.

BASTIANELLI, Emanuele et al. On-line semantic mapping. In: IEEE. 2013 16th International Conference on Advanced Robotics (ICAR). [S.l.: s.n.], 2013. p. 1–6.

BEMELMANS, Roger et al. Socially assistive robots in elderly care: A systematic review into effects and effectiveness. **Journal of the American Medical Directors Association**, Elsevier, v. 13, n. 2, p. 114–120, 2012.

BOUTILIER, Craig; DEAN, Thomas; HANKS, Steve. Decision-theoretic planning: Structural assumptions and computational leverage. **Journal of Artificial Intelligence Research**, v. 11, p. 1–94, 1999.

CANAL, Gerard; ALENYÀ, Guillem; TORRAS, Carme. Adapting robot task planning to user preferences: an assistive shoe dressing example. **Autonomous Robots**, Springer, v. 43, n. 6, p. 1343–1356, 2019.

CANAL, Gerard et al. Probabilistic Planning for Robotics with ROSPlan. In: SPRINGER. ANNUAL Conference Towards Autonomous Robotic Systems. [S.l.: s.n.], 2019. p. 236–250.

CASHMORE, Michael et al. Rosplan: Planning in the robot operating system. In: TWENTY-FIFTH International Conference on Automated Planning and Scheduling. [S.l.: s.n.], 2015.

CROSBY, Matthew et al. Integrating Mission, Logistics, and Task Planning for Skills-Based Robot Control in Industrial Kitting Applications. In: PLANSIG. [S.l.: s.n.], 2016.

DI PAOLA, Donato et al. An autonomous mobile robotic system for surveillance of indoor environments. **International Journal of Advanced Robotic Systems**, SAGE Publications Sage UK: London, England, v. 7, n. 1, p. 8, 2010.

D'INVERNO, Mark; LUCK, Michael; LUCK, Michael M. **Understanding agent systems**. [S.l.]: Springer Science & Business Media, 2004.

DONDRUP, Christian; PAPAIOANNOU, Ioannis; LEMON, Oliver. Petri Net Machines for Human-Agent Interaction. **arXiv preprint arXiv:1909.06174**, 2019.

FARINELLI, Alessandro et al. Assignment of dynamically perceived tasks by token passing in multirobot systems. **Proceedings of the IEEE**, IEEE, v. 94, n. 7, p. 1271–1288, 2006.

FIKES, Richard E; NILSSON, Nils J. STRIPS: A new approach to the application of theorem proving to problem solving. **Artificial intelligence**, Elsevier, v. 2, n. 3-4, p. 189–208, 1971.

GEISSER, Florian; SPECK, David. Prost-DD–utilizing symbolic classical planning in THTS. **IPPC-6 planner abstracts**, 2018.

GEISSER, Florian; SPECK, David; KELLER, Thomas. An Analysis of the Probabilistic Track of the IPC 2018. **WIPC 2019**, p. 27.

GHALLAB, Malik; NAU, Dana; TRAVERSO, Paolo. **Automated Planning: theory and practice**. [S.l.]: Elsevier, 2004.

HARRIS, Catherine Ann. **Online plan modification in uncertain resource-constrained environments**. 2015. Tese (Doutorado) – University of Birmingham.

HOFFMANN, Jörg; BRAFMAN, Ronen. Contingent planning via heuristic forward search with implicit belief states. In: PROC. ICAPS. [S.l.: s.n.], 2005. v. 2005.

HOLGUIN, Mijail Gamarra. **Planejamento probabilístico usando programação dinâmica assíncrona e fatorada**. 2013. Tese (Doutorado) – Universidade de São Paulo.

IOCCHI, Luca et al. A practical framework for robust decision-theoretic planning and execution for service robots. In: TWENTY-SIXTH International Conference on Automated Planning and Scheduling. [S.l.: s.n.], 2016.

JUNIOR, Plinio Thomaz Aquino et al. Hera: Home environment robot assistant. In: II BRAHUR and III Brazilian Workshop on Service Robotics. [S.l.: s.n.], 2019.

KANG, Mun-cheon et al. A robust obstacle detection method for robotic vacuum cleaners. **IEEE Transactions on Consumer Electronics**, IEEE, v. 60, n. 4, p. 587–595, 2014.

KELLER, Thomas; EYERICH, Patrick. PROST: Probabilistic planning based on UCT. In: TWENTY-SECOND International Conference on Automated Planning and Scheduling. [S.l.: s.n.], 2012.

KOCSIS, Levente; SZEPESVÁRI, Csaba. Bandit based monte-carlo planning. In: SPRINGER. EUROPEAN conference on machine learning. [S.l.: s.n.], 2006. p. 282–293.

KUSHMERICK, Nicholas; HANKS, Steve; WELD, Daniel S. An algorithm for probabilistic planning. **Artificial Intelligence**, Elsevier, v. 76, n. 1-2, p. 239–286, 1995.

LIMA, Oscar; VENTURA, Rodrigo; AWAAD, Iman. Integrating classical planning and real robots in industrial and service robotics domains. In: PROCEEDINGS of the 6th Workshop on Planning and Robotics (PlanRob) at ICAPS. [S.l.: s.n.], 2018.

LITTLE, Iain; THIEBAUX, Sylvie et al. Probabilistic planning vs. replanning. In: ICAPS Workshop on IPC: Past, Present and Future. [S.l.: s.n.], 2007.

LITTMAN, Michael L. Markov games as a framework for multi-agent reinforcement learning. In: MACHINE learning proceedings 1994. [S.l.]: Elsevier, 1994. p. 157–163.

LUCAS MARTÍNEZ, Néstor et al. Survey of Mission Planning and Management Architectures for Underwater Cooperative Robotics Operations. **Applied Sciences**, Multidisciplinary Digital Publishing Institute, v. 10, n. 3, p. 1086, 2020.

MARTÍNEZ, David et al. Relational reinforcement learning for planning with exogenous effects. **The Journal of Machine Learning Research**, JMLR. org, v. 18, n. 1, p. 2689–2732, 2017.

MATAMOROS, Mauricio; SEIB, Viktor; PAULUS, Dietrich. Trends, Challenges and Adopted Strategies in RoboCup@ Home. **arXiv preprint arXiv:1903.02516**, 2019.

MATAMOROS, Mauricio et al. RoboCup@ Home: Summarizing achievements in over eleven years of competition. In: IEEE. 2018 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC). [S.l.: s.n.], 2018. p. 186–191.

MORRIS, Robert et al. Autonomous search-detect-track for small uavs. **IntEx 2017**, p. 19, 2017.

MÜLLER, Jörg P. Architectures and applications of intelligent agents: A survey. **The Knowledge Engineering Review**, Cambridge University Press, v. 13, n. 4, p. 353–380, 1999.

NIEMUELLER, Tim; HOFMANN, Till; LAKEMEYER, Gerhard. CLIPS-based execution for PDDL planners. In: ICAPS Workshop on Integrated Planning, Acting and Execution (IntEx). [S.l.: s.n.], 2018.

NOGUEIRA, Vinicius Veloso Eleutério. Planejamento de Missão para VANTs em Ambientes Estocásticos, 2018.

PALAMARA, Pier Francesco et al. A robotic soccer passing task using Petri net plans. In: PROCEEDINGS of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers. [S.l.: s.n.], 2008. p. 1711–1712.

PEDNAULT, Edwin PD. Formulating multiagent, dynamic-world problems in the classical planning framework. In: REASONING about actions & plans. [S.l.]: Elsevier, 1987. p. 47–82.

QUIGLEY, Morgan et al. ROS: an open-source Robot Operating System. In: KOBE, JAPAN, 3.2. ICRA workshop on open source software. [S.l.: s.n.], 2009. v. 3, p. 5.

RUSSELL, Stuart J; NORVIG, Peter. **Artificial intelligence: a modern approach**. [S.l.]: Malaysia; Pearson Education Limited, 2016.

SANELLI, Valerio et al. Short-term human-robot interaction through conditional planning and execution. In: TWENTY-SEVENTH International Conference on Automated Planning and Scheduling. [S.l.: s.n.], 2017.

SANNER, Scott. Relational dynamic influence diagram language (rddl): Language description. **Unpublished ms. Australian National University**, v. 32, 2010.

SHPIEVA, Elizaveta; AWAAD, Iman. Integrating task planning, execution and monitoring for a domestic service robot. **it-Information Technology**, De Gruyter Oldenbourg, v. 57, n. 2, p. 112–121, 2015.

SMITH, Trey; SIMMONS, Reid. **Probabilistic planning for robotic exploration**. 2007. Tese (Doutorado) – Carnegie Mellon University, The Robotics Institute.

SPECK, David; DORNHEGE, Christian; BURGARD, Wolfram. Shakey 2016—How much does it take to redo shakey the robot? **IEEE Robotics and Automation Letters**, IEEE, v. 2, n. 2, p. 1203–1209, 2017.

TOMATIS, Nicola et al. Building a fully autonomous tour guide robot: Where academic research meets industry. In: ISR. PROCEEDINGS of the 33rd International Symposium on Robotics (ISR). [S.l.: s.n.], 2002. p. 109–134.

WILSON, Garrett et al. Robot-enabled support of daily activities in smart home environments. **Cognitive Systems Research**, Elsevier, v. 54, p. 258–272, 2019.

WOOLDRIDGE, Michael. **An introduction to multiagent systems**. [S.l.]: John Wiley & Sons, 2009.

YOUNES, Håkan LS; LITTMAN, Michael L. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. **Techn. Rep. CMU-CS-04-162**, v. 2, p. 99, 2004.

ZIPARO, Vittorio A et al. Petri net plans. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 23, n. 3, p. 344–383, 2011.