



COPPE/UFRJ

ARQUITETURA HÍBRIDA DE UM ROBÔ MÓVEL GUIADO POR TRILHOS

Renan Salles de Freitas

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Ramon Romankevicius

Rio de Janeiro

Março de 2016

ARQUITETURA HÍBRIDA DE UM ROBÔ MÓVEL GUIADO POR TRILHOS

Renan Salles de Freitas

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Aprovada por:

Prof. Ramon Romankevicius, D.Sc.

Prof. , D.Sc.

Prof. , Ph.D.

Prof. , D.Sc.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2016

Freitas, Renan Salles de

Arquitetura híbrida de um robô móvel guiado por
trilhos/Renan Salles de Freitas. – Rio de Janeiro:
UFRJ/COPPE, 2016.

XIV, 116 p.: il.; 29, 7cm.

Orientador: Ramon Romankevicius

Dissertação (mestrado) – UFRJ/COPPE/Programa de
Engenharia Elétrica, 2016.

Referências Bibliográficas: p. 110 – 116.

1. Controle de Missão. 2. Arquitetura robótica.
 3. Máquina de estado. 4. Paradigmas da robótica.
 5. ROS. 6. SMACH. I. Romankevicius, Ramon.
- II. Universidade Federal do Rio de Janeiro, COPPE,
Programa de Engenharia Elétrica. III. Título.

à minha família

Agradecimentos

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ARQUITETURA HÍBRIDA DE UM ROBÔ MÓVEL GUIADO POR TRILHOS

Renan Salles de Freitas

Março/2016

Orientador: Ramon Romankevicius

Programa: Engenharia Elétrica

Sistemas autônomos inteligentes são cada vez mais explorados no meio acadêmico e industrial, por serem um desafio multidisciplinar, financeiramente atrativos, robustos com alta confiabilidade, melhorarem as condições de vida e trabalho do ser humano, e minimizarem possíveis falhas humanas em sistemas de alto risco. A tecnologia de software e hardware para construção destes sistemas é desenvolvida desde 1950, e até hoje três paradigmas já foram idealizados para responder a pergunta: qual a maneira correta de construir um sistema autônomo? A compreensão e entendimento das diversas arquiteturas robóticas e a avaliação de casos, suas vantagens e desvantagens permitem a análise crítica dos sistemas, e propostas de melhoramentos. O estudo de arquiteturas robóticas e como suas instâncias são usadas para uma determinada aplicação, possibilita o aprendizado das diferentes formas que os componentes e ferramentas associados a um paradigma podem ser usados para a construção de uma inteligência artificial robótica. O objetivo deste trabalho é a implementação de uma arquitetura robótica para uma classe de robôs móveis com aplicação de inspeção. O estudo de caso para esta dissertação é a DORIS, robô *offshore* para inspeção em plataformas de petróleo. Portanto, é desenvolvido um sistema autônomo inteligente para a DORIS, seguindo uma metodologia e critérios

de avaliação conforme o estado da arte. Resultados e testes mostraram o desempenho positivo da arquitetura e uma análise comparativa evidenciou a implementação intuitiva, modular e flexível da solução.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

HYBRID ARCHITECTURE OF A RAIL GUIDED MOBILE ROBOT

Renan Salles de Freitas

March/2016

Advisor: Ramon Romankevicius

Department: Electrical Engineering

Intelligent autonomous systems research grows rapidly in academia and industry, as they are a multi-disciplinary challenge, financially attractive, robust with high reliability, they improve human living conditions and labor, and minimize possible human errors in high-risk systems. Since 1950, the software and hardware technology for these systems is developed, and three paradigms were idealized to try to answer the question: what is the correct way to build an intelligent autonomous system? The comprehension of the various robotic architectures and robotic cases, and their advantages and disadvantages allow analysis, and improvement proposals. The study of robotic architectures and how they are used allow the understanding of how different components and tools associated with a paradigm can build an artificial intelligence robot. The objective of this work is the implementation of a robotic architecture for mobile inspection robots and similar applications. The case study for this thesis DORIS, a mobile robot for remote supervision, diagnosis, and data acquisition on offshore facilities. Therefore, an autonomous system is developed for DORIS, following the methodology and criteria evaluation in accordance with the state of the art. Tests and results showed the positive performance of the proposed architecture and a comparative analysis highlighted the intuitive, modular and flexible implementation of the solution.

Sumário

Lista de Figuras	xi
Lista de Tabelas	xiv
1 Introdução	1
1.1 Motivação	3
1.1.1 Sistemas autônomos	3
1.1.2 Arquiteturas	4
1.2 Objetivo	5
1.3 Metodologia	5
1.4 Organização da tese	6
2 Revisão Bibliográfica	8
2.1 Conceitos base	8
2.2 Paradigma hierárquico/deliberativo	10
2.2.1 Robôs deliberativos	11
2.2.2 Arquiteturas deliberativas	12
2.2.3 Análise crítica	19
2.3 Paradigma reativo	21
2.3.1 Robôs reativos	23
2.3.2 Arquiteturas reativas	24
2.3.3 Análise crítica	31
2.4 Paradigma híbrido ou deliberativo/reactivo	32
2.4.1 Robôs híbridos	33
2.4.2 Arquitetura híbrida	33
2.4.3 Análise crítica	43

2.5	Robotic Development Environments	43
2.5.1	Camada Funcional	44
2.5.2	Camada Executivo	49
2.5.3	Camada Planejador	56
3	Arquitetura proposta	59
3.1	Robô DORIS	59
3.2	A arquitetura robótica implementada	62
3.2.1	Implementação da camada Planejador	63
3.2.2	Implementação da camada Funcional	71
3.2.3	Implementação da camada Executivo	77
4	Resultados e Discussões	88
4.1	Metodologia para avaliação das camadas	88
4.2	Testes da implementação da camada Funcional	90
4.2.1	Testes de planejamento de velocidades	92
4.3	Testes da implementação da camada Executivo	95
4.4	Testes da implementação da camada Planejador	100
5	Conclusões e trabalhos futuros	106
5.1	Trabalhos futuros	108
Referências Bibliográficas		110

Lista de Figuras

2.1	Paradigma hierárquico tradicional.	11
2.2	Shakey robot	12
2.3	HILARE	12
2.4	Manipulador robótico atual	13
2.5	Arquitetura de Albus para sistemas deliberativos.	15
2.6	Arquitetura NASREM.	16
2.7	Arquitetura de Saridis	17
2.8	Arquitetura do Veículo MARIUS	20
2.9	Comparativo da arquitetura deliberativa com o ser humano.	21
2.10	Projeto da empresa Google 2011 - para o desenvolvimento de um carro autônomo.	21
2.11	Arquitetura para sistema de controle de robôs móveis por Brooks	24
2.12	A tartaruga de Grey Walter.	25
2.13	Veículos de Braitenberg.	25
2.14	Robô comercial Roomba da iRobot.	25
2.15	Camadas de controle de Brooks	27
2.16	Nível 0 e 1 de controle do sistema	28
2.17	Módulo básico da arquitetura de subsunção (AFSM).	29
2.18	Analogia de sistemas reativos com o ser humano.	32
2.19	Robô George.	34
2.20	O AUV Phoenix.	34
2.21	Robô Curiosity, o robô da NASA explorador do planeta Marte.	34
2.22	Arquitetura AuRA.	38
2.23	Arquitetura do Robô Phoenix de Healey	41
2.24	Arquitetura do Robô Stanley	42

2.25	Analogia de sistemas reativos com o ser humano.	44
2.26	Tipos de comunicação so sistema ROS.	48
2.27	Arquitetura do ambiente de desenvolvimento ROS.	48
2.28	Container <i>SM_TOP</i> que possui dois estados e transições de dados. . .	55
3.1	A DORIS e o trilho.	61
3.2	A DORIS no CENPES.	62
3.3	Arquitetura da camada Planejador.	65
3.4	Organização da camada Funcional CLARAty	71
3.5	Camada Funcional da DORIS	75
3.6	Arquitetura da camada Executivo.	78
3.7	Exemplo do módulo básico para a missão simples GOTO	80
3.8	Estado CAMERA do processo reativo específico OBSTACLE_DETECTION	82
3.9	Estado EPOS do processo reativo específico OBSTACLE_DETECTION	82
3.10	Estados do processo reativo específico OBSTACLE_DETECTION	82
3.11	Tarefas da missão simples GOTO	84
3.12	Estados da tarefa Localization	84
3.13	Estados da tarefa Motion_Position_Control	85
3.14	Estados da tarefa RECORD_TYPE	85
4.1	Teleoperação da DORIS.	92
4.2	Interface de controle da DORIS.	92
4.3	representação gráfica do plano de velocidades do robô para uma missão de GOTO(10,'fast').	94
4.4	Processos em execução durante missão GOTO	97
4.5	Transições da missão simples GOTO no terminal.	98
4.6	Transições da missão simples GOTO no smach_viewer.	99
4.7	Ao receber uma informação de nível de bateria inferior a 5%, Charge aborta a missão GOTO (terminal).	99
4.8	Ao receber uma informação de nível de bateria inferior a 5%, Charge aborta a missão GOTO (smach_viewer).	99

4.9	Ao receber uma informação de recurso indisponível, Epos aborta a missão GOTO (terminal).	100
4.10	Método <i>execute</i> da missão INSPECTION	103
4.11	Missões e processos na execução da missão complexa INSPECTION .	105

Lista de Tabelas

2.1	Tabela comparativa de arquitetura deliberativa e reativa	32
2.2	Resumo da arquitetura AuRA	37
2.3	Resumo da arquitetura de três camadas	40
4.1	Matriz de velocidades máximas em cada trecho do trilho do ponto 0 ao 10.	93
4.2	Mapa das seções do trilho	94

Capítulo 1

Introdução

É certo afirmar que a robótica e sistemas autônomos são cada vez mais explorados no meio acadêmico e industrial, por serem um desafio multidisciplinar, financeiramente atrativos, e melhorarem as condições de vida e trabalho do ser humano. Este trabalho é um estudo de arquiteturas robóticas de sistemas inteligentes, isto é, sistemas mecânicos autônomos [1].

Ser “inteligente” implica que o sistema não executa ações simples e repetitivas, ou seja, é o oposto de robôs em linhas de produção, como manipuladores industriais. “Sistema mecânico” implica que o robô não é um simples computador, mas possui partes mecânicas e pode interagir fisicamente com o mundo em que está inserido, como mover-se, e modificá-lo. Por último, “autônomo” indica que o robô pode operar independentemente, sem a necessidade de um operador, pode se adaptar a mudanças do mundo em que está inserido ou de si mesmo, e continuar a alcançar seus objetivos.

Há diversas obras da ficção que são ótimos exemplos de sistemas mecânicos autônomos inteligentes, como o “Exterminador do Futuro”, “Ava” de Ex-Machina, e “Wall-E”. Sistemas desta complexidade ainda não são uma realidade, mas há esforços na literatura de sistemas complexos humanóides, como o robô Asimo [2], e robôs móveis autônomos e teleoperados espaciais, como o Curiosity da NASA. Os sistemas autônomos requerem integração multidisciplinar, como engenharia mecânica, engenharia elétrica, engenharia eletrônica, engenharia de controle, ciência da computação, *Machine learning*, e outras. A literatura para cada subsistema é ampla, mas há, ainda, o sistema de integração e o software que transforma o sistema em

um robô inteligente e autônomo, a arquitetura robótica, escopo deste trabalho.

As arquiteturas robóticas começaram a ser idealizadas desde 1960, junto com os estudos emergentes de inteligência artificial, e percorreram, durante anos, paradigmas em sua construção. Paradigma é uma filosofia ou conjunto de premissas ou técnicas que caracteriza uma abordagem a uma classe de problemas. Não há paradigma correto ou errado, e o mesmo pode se afirmar para as arquiteturas robóticas, mas alguns problemas são melhor resolvidos em um paradigma em relação a outro. Por exemplo, em problemas de cálculo, alguns são resolvidos facilmente em uma abordagem de coordenadas cartesianas, outros em uma abordagem de coordenadas polares. Aplicar o paradigma correto a um determinado problema pode facilitar sua solução, logo o estudo dos paradigmas da inteligência artificial robótica é chave para o desenvolvimento de arquiteturas adequadas aos diversos problemas na construção de sistemas autônomos. Dessa forma, os paradigmas e arquiteturas da robótica são explicitados a fim de se desenvolver uma proposta de arquitetura para robôs móveis.

A classe de robôs que mais demanda sistemas inteligentes são robôs móveis. Esta classe apresenta diversos desafios, como a locomoção (atuadores) e mecânica, percepção (sensores) e processamento, teoria de controle, localização, planejamento de trajetória e navegação, entre outros. Os desafios podem ser explorados individualmente, e a literatura possui diversos estudos individuais em relação à localização ou sistemas de locomoção, por exemplo. A integração de todas as soluções individuais de uma determinada maneira compõe o sistema, e o estudo de sistemas autônomos é a investigação de como integrá-los e fornecer um software de adaptação às possíveis mudanças do meio e de si. Neste trabalho, são analisadas arquiteturas robóticas de robôs autônomos subaquáticos (AUV), veículos autônomos não tripulados (UGV), e outros sistemas.

A compreensão e entendimento das diversas arquiteturas robóticas e a avaliação de casos bem sucedidos, suas vantagens e desvantagens permitem a análise crítica dos sistemas e a proposta de modificações e melhoramentos. O autor busca propor uma implementação de arquitetura robótica para uma subclasse de sistemas, generalizar para outras subclasses. O robô DORIS¹ será o estudo de caso, um veículo de inspeção

¹This work is supported primarily by Petrobras S.A. and Statoil Brazil Oil & Gas Ltda under contract COPPETEC 0050.0079406.12.9 (ANP-Brazil R&D Program), and in part by the Brazilian research agencies CNPq and FAPERJ.

guiado por trilhos. Finalmente, é criada uma metodologia geral para a avaliação de arquiteturas de mesma complexidade e são realizadas simulações para a análise dos critérios.

1.1 Motivação

A motivação deste trabalho é baseada em duas perguntas principais: por que utilizar sistemas autônomos? Por que desenvolver uma arquitetura robótica para sistemas autônomos? As subseções a seguir tentam motivar esses aspectos.

1.1.1 Sistemas autônomos

A utilização de robôs na indústria é uma realidade desde 1962, quando o primeiro robô industrial faz soldagem na fábrica da General Motors. Desde então, robôs assumiram diferentes aplicações na indústria e suas principais razões para uso são:

- Redução de custo em operação;
- Quando o ser humano não é capaz de executar a tarefa, como manipulação de semicondutores;
- Quando o manuseio do produto por um ser humano não é higiênico, como manipulação de medicamentos e alimentos;
- Quando o produto faz mal à saúde do ser humano, como o manuseio de substâncias químicas ou radioativas;
- Quando o maquinário pode colocar a vida do operador em risco, como prensas;
- Quando a tarefa pode prejudicar a saúde do ser humano por repetição;
- Quando é requerida precisão extrema e grau de eficiência impossíveis a um ser humano;
- Quando o ambiente é impróprio para o ser humano, como ambientes explosivos, e outros.

Os robôs, aos poucos, começaram a sair das fábricas e usados em aplicações que exigiram o desenvolvimento de uma nova classe de robôs, os robôs móveis, o que acelerou linhas de pesquisa na área da telerobótica. A teleoperação indica a operação de uma máquina à distância, e requer o conjunto: operador, robô e controle remoto.

As aplicações de robôs móveis teleoperados são diversas, como os veículos operados remotamente (ROVs) para exploração do fundo do mar, robôs para desarmamento de bombas, e robôs espaciais, como o Curiosity da NASA. Estes sistemas teleoperados são também chamados de sistemas semi-autônomos, já que o nível de complexidade da programação de baixo nível do robô é impossível de ser controlada diretamente por um operador. O ser humano pode realizar a localização e as atividades cognitivas, mas o esquema de controle embarcado ao robô é o que provê o controle de movimento. Por exemplo, o veículo subaquático descrito em [3] controla os seis propulsores (atuadores) de maneira autônoma e estabiliza o robô em águas turbulentas e com correnteza, enquanto o operador escolhe posições objetivo para o robô.

Os níveis de autonomia podem ser extrapolados até veículos completamente autônomos, por exemplo, como em AUVs, o carro autônomo da Google, e robôs já comerciais, como o Roomba da iRobot para limpeza doméstica. O desenvolvimento de sistemas autônomos absorve as motivações já citadas, mas os fatores que mais estimulam a adoção destes sistemas, como exposto em [4], são a confiabilidade do sistema, a confiança do ser humano no sistema, treinamento e o conhecimento dos modos de falha. O sistema de piloto automático de aviões, por exemplo, levou tempo até ser aceito, porém o número de variáveis e o risco motivaram o desenvolvimento de sistemas robustos, confiáveis e autônomos.

O robô DORIS, caso de estudo deste trabalho, é um robô de inspeção para plataformas de petróleo, um ambiente de grande perigo com risco de explosão, condições climáticas desfavoráveis, exigindo, portanto, um grande grau de autonomia para proteção do robô e das pessoas que circulam o mesmo ambiente de trabalho.

1.1.2 Arquiteturas

Para entendermos a importância de uma arquitetura, considere construir uma casa ou um carro. Não há um projeto certo para uma casa, porém a maioria das casas

possuem os mesmos componentes: cozinha, banheiro, paredes, chão, portas e outros. Isto é similar ao exemplo de carros. Produzidos por diferentes fabricantes, todos os tipos de motores de combustão interna possuem os mesmos componentes básicos, porém os carros parecem diferentes.

Voltando ao conceito de paradigma (capítulo 1), a combustão interna e carros elétricos são dois paradigmas, e dentro da comunidade dos veículos à combustão cada fabricante possui a sua própria arquitetura. Os fabricantes podem realizar algumas pequenas modificações para carros do tipo sedan, conversível, sport e etc, para suprimir algumas opções desnecessárias, mas cada estilo de carro é uma instância particular da arquitetura do fabricante.

Os exemplos acima são analogias semelhantes à construção de robôs. Há diversas maneiras de organizar seus componentes, mesmo que o projeto siga o mesmo paradigma. Dessa forma, o ponto principal é: ao estudarmos arquiteturas robóticas e como suas instâncias são usadas para uma determinada aplicação, podemos aprender as diferentes formas que os componentes e ferramentas associados a um paradigma podem ser usados para a construção de uma inteligência artificial robótica. O estudo de sistemas inteligentes requer, portanto, o estudo das arquiteturas robóticas.

1.2 Objetivo

O objetivo deste trabalho é a implementação de uma arquitetura robótica para uma classe de robôs móveis com aplicação de inspeção. O estudo de caso para esta dissertação é a DORIS, robô *offshore* para inspeção em plataformas de petróleo. Portanto, será desenvolvido um sistema autônomo inteligente para a DORIS, seguindo uma metodologia e critérios de avaliação conforme o estado da arte.

1.3 Metodologia

A metodologia para alcançar o objetivo proposto passa pelas seguintes etapas: estudo dos paradigmas da robótica; estudo de arquiteturas robóticas e suas aplicações; pesquisa do estado da arte de ambientes de desenvolvimento robóticos; análise crítica e solução proposta; desenvolvimento da metodologia de avaliação; testes e resultados.

Os três paradigmas da robótica e as arquiteturas robóticas são bem contemplados na literatura e há diversos textos acadêmicos que os exploram: paradigma deliberativo, reativo, e deliberativo/reactivo. O estudo da evolução dos paradigmas é o primeiro passo para a construção de um sistema autônomo para os robôs. A metodologia utilizada nesta revisão bibliográfica buscou estudar as arquiteturas e aplicações pioneiras, as arquiteturas de transição (intermediárias cronologicamente), e as mais atuais.

Após explorar os paradigmas e arquiteturas robóticos e definir a mais adequada para o estudo de caso, faz-se necessário investigar os ambientes de desenvolvimento robóticos (RDEs), ferramentas que facilitam a implementação da arquitetura. Há diversos disponíveis, portanto suas vantagens e desvantagens devem ser avaliadas. O estudo de RDEs pode proporcionar a utilização de um extenso repositório criado pela academia, facilitando e divulgando novos pacotes de software para robôs.

A análise crítica das arquiteturas e os RDEs, visando a aplicação, possibilita a implementação da arquitetura robótica proposta. O estudo de caso é introduzido e a solução é desenvolvida tomando-o como exemplo. É criada uma metodologia de avaliação da arquitetura, com base em alguns roboticistas, e critérios para avaliar a implementação, por simulações ou testes exaustivos no robô.

1.4 Organização da tese

O capítulo 2 faz uma revisão bibliográfica, apresentando conceitos básicos da literatura necessários para o entendimento da dissertação, desenvolvendo novos conceitos pela visão do autor, estudando os paradigmas da robótica, a evolução das arquiteturas robóticas, e casos de robôs com análises críticas.

O capítulo 3 faz uma proposta de implementação de arquitetura robótica, apresenta o robô DORIS, estudo de caso deste trabalho, e detalha a implementação da arquitetura para o robô.

O capítulo 4 propõe uma metodologia de avaliação, realiza simulações e testes, e analisa a implementação realizada para a arquitetura robótica do capítulo anterior pelos critérios estabelecidos.

Por fim, o capítulo 5 conclui o estudo, faz uma análise crítica do desenvolvimento

e propõe trabalhos futuros no tópico.

Capítulo 2

Revisão Bibliográfica

A modelagem de robôs de acordo com suas principais funcionalidades e o desenvolvimento de novas arquiteturas são o âmago no estudo de autonomia e controle de missão de robôs móveis. Dessa forma, arquitetura robótica e controle de missão são conceitos relacionados, e, portanto, o cerne desta pesquisa bibliográfica.

Neste capítulo, são apresentados os fundamentos teóricos necessários para o entendimento desta dissertação. O objetivo deste levantamento bibliográfico é apresentar alguns dos principais trabalhos e pesquisas científicos sobre arquiteturas e sistemas de controle de missão de robôs móveis. Por fim, o leitor é direcionado para os ambientes de software desenvolvidos para robôs autônomos (*Robotic Development Environments - RDEs*).

2.1 Conceitos base

Faz-se necessário a definição de alguns conceitos base para o entendimento desta dissertação e para a fase de implementação da arquitetura.

O conceito de arquitetura para robôs é definido de diferentes formas na literatura. Em [5], arquitetura de robô é relacionada com arquitetura de software, em uma adaptação à arquitetura de computadores de [6], e definida como: arquitetura de robô é a disciplina dedicada ao projeto de robôs altamente específicos e individuais a partir de uma coleção de blocos comuns de softwares. Em [7], a definição aborda sistemas de controle: uma arquitetura fornece uma maneira principal de organizar um sistema de controle, contudo, a arquitetura também impõe restrições sobre a

forma como o problema de controle pode ser resolvido. Já em [8], o autor tenta associar a arquitetura de software com os componentes de hardware (processadores) para compor a arquitetura robótica.

O sistema robótico é composto por diversos elementos de hardware e software que são interdependentes e necessários para o funcionamento do sistema. Como o propósito deste trabalho é o desenvolvimento de uma arquitetura robótica para o robô DORIS, sendo considerados os aspectos físicos, lógicos e a aplicabilidade do robô, entende-se que:

Definição: Uma **arquitetura robótica** descreve uma maneira de se construir o controle inteligente do robô, os módulos do sistema, como estes módulos interagem entre si e seus elementos de hardware associados, visando sua aplicação. Os elementos de hardware e a aplicação assumem um papel de grande importância durante o desenvolvimento dessas arquiteturas, por exemplo como um fator limitador.

Uma definição equivalente de **arquitetura robótica**, porém mais objetiva é:

Definição 2: Uma **arquitetura robótica** é uma arquitetura que pode ser avaliada pelos critérios de Arkin: **Suporte a paralelismo**, **Hardware targetability**, **Niche targetability**, **Suporte a modularidade**, **Robustez**, **Tempo de desenvolvimento**, **Flexibilidade em tempo de execução** e **Desempenho em executar tarefas** (detalhes em seção 4.1).

De acordo com [9], os componentes básicos de uma arquitetura para robôs são classificados em três grupos: *Percepção*, que envolve as atividades de interpretação e integração dos sensores; *Planejamento* de tarefas, sincronização, e o monitoramento da execução de todas as atividades do robô; e *Atuação*, que envolve as atividades de execução dos movimentos, ações do robô e controle dos atuadores.

As três primeiras seções desta pesquisa bibliográfica abordam os paradigmas da robótica, isto é, as três arquiteturas de operação de um sistema robótico: paradigma hierárquico/deliberativo (SPA - *Sense, Plan and Act*); paradigma reativo; e paradigma híbrido deliberativo/reactivo. As seções apresentam e exemplificam as arquiteturas pela ótica de diversos autores, e são comparadas e analisadas.

O controle de missão (*Mission Control System* - MCS) ou planejamento de missão (*Mission Planning*) ou planejamento de tarefas (*Task Planning*) de robôs faz parte da arquitetura robótica, e pode ser desenvolvido para os três tipos de arquiteturas.

Em [10], o conceito é bem introduzido como: controle de missão é um sistema que permite ao operador definir as missões de um veículo em linguagem de alto nível; provê ferramentas adequadas para converter planos em Progamas de Missões que podem ser verificados e executados em tempo real; e permite ao operador saber o estado da missão enquanto esta é executada, e modificá-la se for necessário. Em [11], o conceito de planejamento de missão é ampliado para múltiplos robôs: planejamento de missão é o processo de determinar o que cada robô deve fazer para alcançar, de uma maneira conjunta, os objetivos da missão, em um ambiente dinâmico.

Neste trabalho, entende-se como controle de missão de sistemas robóticos:

Definição: o controle de missão de robôs é o componente da arquitetura robótica que permite ao operador definir as missões de um veículo, exerce o papel de traduzir os comandos de missão do usuário ao robô, provê feedback ao operador (*execution monitoring*), e contém as diretivas do robô.

O controle de missão faz parte da arquitetura robótica, portanto as seções que seguem buscam, em cada arquitetura, destacar de forma exemplificada alguns controles de missão.

A DORIS é um robô móvel com aplicação de inspeção em um ambiente dinâmico. Apresenta missões semelhantes a um AUV, apesar de sua dinâmica ser mais simplificada, já que este se move ao longo de um trilho. Entretanto, como a aplicação (inspeção) é igual, os hardwares são equivalentes por ser um robô móvel com vários sensores, e alguns desafios são comuns. É de se esperar que a arquitetura robótica possa apresentar muitas semelhanças. Desta forma, durante a apresentação das arquiteturas, será sempre destacada uma arquitetura de AUV ou UGV de aplicação semelhante.

2.2 Paradigma hierárquico/deliberativo

Em meados do século XX, são realizados os primeiros estudos de robôs autônomos, juntamente com o aparecimento da Inteligência Artificial (IA). Em um sistema robótico, a IA clássica consiste em um modelo centralizado que coleta informações usando sensores, cria um modelo do ambiente, planeja o próximo movimento e executa a ação. São sistemas do tipo *Sense, Plan and act* (SPA) ou *top-down*

(hierárquico), como na decomposição tradicional, figura 2.1.

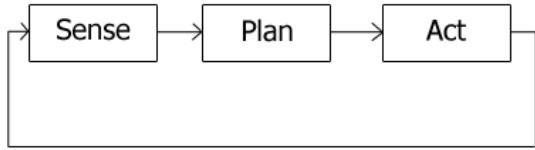


Figura 2.1: Paradigma hierárquico tradicional.

De acordo com Marvin Minsky, uma máquina (robô) deveria tender a criar, por si só, um modelo abstrato do ambiente em que está inserido (define-se *mundo*). Caso fosse dada uma tarefa, o robô poderia, primeiramente, explorar soluções dentro de seu modelo abstrato e, então, experimentá-las externamente. Seria como realizar uma simulação interna e, caso funcionasse, executá-la.

2.2.1 Robôs deliberativos

Entre 1966 e 1972, Charles Rosen e Nils Nilsson da Universidade de Stanford criaram o Shakey, primeiro robô móvel autônomo (figura 2.2). Foi desenvolvida uma inteligência artificial chamada STRIPS (*problem solver*). Este sistema é um planejador de trajetórias que armazena as informações do ambiente (mapas e obstáculos) de maneira simbólica, e, se dada uma tarefa de deslocamento (*goto*), é realizada uma busca lógica pelo sistema.

Em 1977, começou a ser desenvolvido o projeto HILARE (figura 2.3), no Laboratoire d'Automatique et d'Analyse des Systèmes (LAAS), Toulouse, França. O robô possuía sensores como câmera, ultrassons e laser para medir distância, sendo possível atualizar o seu mundo com acurácia. Seu mundo era representado por modelos geométricos e um modelo relacional que expressava a conectividade dos quartos e corredores (simbólico) [12].

Também em 1977, o Stanford Cart foi criado por Moravec para navegação e desvio de obstáculos [13]. Os obstáculos eram identificados pelo robô durante a operação e representados em seu mundo interno como esferas. O robô possuía uma segunda representação do mundo simbólica por grafos.

Em 1969, Victor Scheinman, Universidade de Stanford [14], inventou o primeiro manipulador robótico totalmente elétrico de seis elos e com solução completa e integrada de cinemática inversa. Isto é, dado um ponto qualquer pertencente ao espaço de trabalho do manipulador, este calcula o ângulo das juntas de forma que o efetuador alcance o ponto especificado. Isso permitiu que o manipulador percorresse trajetórias arbitrárias. Até os dias atuais, 2015, é ampla a utilização de manipuladores industriais. A sofisticação destes sistemas já possibilita que estes armazenem todo o conhecimento do mundo e executem tarefas autônomas (figura 2.4).

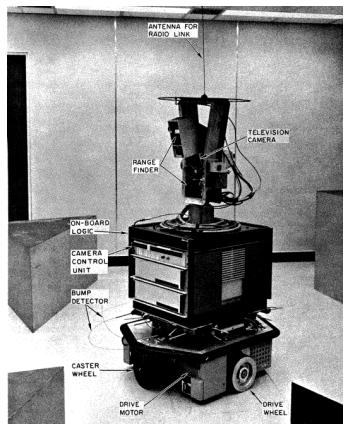


Figura 2.2: Shakey robot

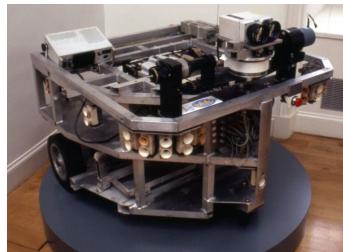


Figura 2.3: HILARE

2.2.2 Arquiteturas deliberativas

As arquiteturas deliberativas são sistemas hierárquicos com a lógica *SPA*. São utilizados em sistemas robóticos até hoje, quando a aplicação favorece seu uso e o poder computacional não é uma restrição. Destacam-se os modelos de Albus, NASREM, o *Intelligent Mobile Robot System*, e o MARIUS (AUV).



Figura 2.4: Manipulador robótico atual

Modelo de Albus

Albus foi o pioneiro e mais influente autor de teorias em arquiteturas deliberativas, [15]. Sua grande contribuição foi a formalização e definição de diversos termos amplamente utilizados em automação e controle. Dentre outros, destaca-se o teorema de que há quatro sistemas que compõem a inteligência: processamento de sensores, modelo do mundo, geração de comportamentos, e julgamento de valor.

- Atuadores: as saídas de um sistema inteligente são produzidas por atuadores, como mover, posicionar braços, pernas, mãos, olhos e etc. Analogamente, os “atuadores” da natureza naturais são os músculos e as glândulas, já os atuadores de máquinas são motores, pistões e válvulas.
- Sensores: são as entradas de um sistema inteligente, como sensores de força, torque, posição, velocidade, vibração, acústico, gases, temperatura e muitos outros. Monitoram o mundo e o estado interno do sistema, e provêem dados ao sistema de processamento sensorial.
- Processamento sensorial: sistema que compara novas observações com a expectativa interna do modelo do mundo. Integra e armazena as diferenças e semelhanças encontradas, a fim de reconhecer padrões, objetos e relações no mundo.
- Modelo do mundo: é a melhor estimativa que o sistema inteligente possui do mundo, e atualizado pelo processamento sensorial. É um banco de dados com todo o conhecimento do mundo e contém uma capacidade de simulação que

gera expectativas e predições. O modelo do mundo pode prover informações do passado, presente e prever estados futuros. Os dados são importantes para: o gerador de comportamentos escolher o plano adequado para execução das ações; o processamento sensorial fazer correlações, comparação de modelos, e reconhecimento de objetos, estados e eventos; e o sistema de julgamento de valor computar valores de custo, benefício, risco, incerteza, importância e outros.

- Julgamento de valor: este é o sistema que determina o que é bom ou ruim, importante ou trivial, certo ou improvável. Computa custos, riscos e benefícios de situações observadas e atividades planejadas.
- Gerador de comportamentos: elemento que seleciona objetivos e planos, executa e monitora ações, e modifica planos existentes quando alguma situação do mundo exigir. Tarefas são decompostas em subtarefas, e subtarefas são sequências de objetivos. A ordem lógica de funcionamento é: o gerador de comportamentos cria planos, o modelo do mundo predita o resultado do plano, e o julgamento de valor avalia os resultados. O gerador de comportamento seleciona o plano com a avaliação mais alta.

As relações entre os elementos do sistema inteligente estão representados na figura 2.5. Esses elementos e suas relações possibilitaram a criação de diversas arquiteturas.

Vale ressaltar que, nesta arquitetura, o *gerador de comportamentos* faz o papel do controle de missão, porém não de maneira completa, já que a interação com o usuário ainda é precária.

NASREM

O NASREM [16] foi uma arquitetura utilizada pela NASA e possuía uma arquitetura com seis níveis de funcionalidade (figura 2.6):

1. Servo: provê o controle dos atuadores do robô (posição, velocidade e etc).
2. Primitiva: determina as primitivas de movimento para gerar trajetórias suaves.
3. Movimento elementar: define e planeja trajetórias livres de colisões.



Figura 2.5: Arquitetura de Albus para sistemas deliberativos.

4. Tarefa: converte ações desejadas de um objeto em sequências de movimentos elementares.
5. Compartimento de serviços: converte ações de grupos de objetos em tarefas de um objeto.
6. Missão: decompõe o plano de missão em alto nível em compartimento de serviços.

Vale ressaltar que, no modelo NASREM, o operador tem acesso a qualquer nível hierárquico do robô e pode tomar o controle do robô para si, além de poder substituir as entradas de sensores, modelo do mundo e outros. Dessa forma, o nível de autonomia do robô pode ser desenvolvido de forma incremental.

A arquitetura hierárquica proposta em NASREM permite modularidade e propõe uma metodologia de software.

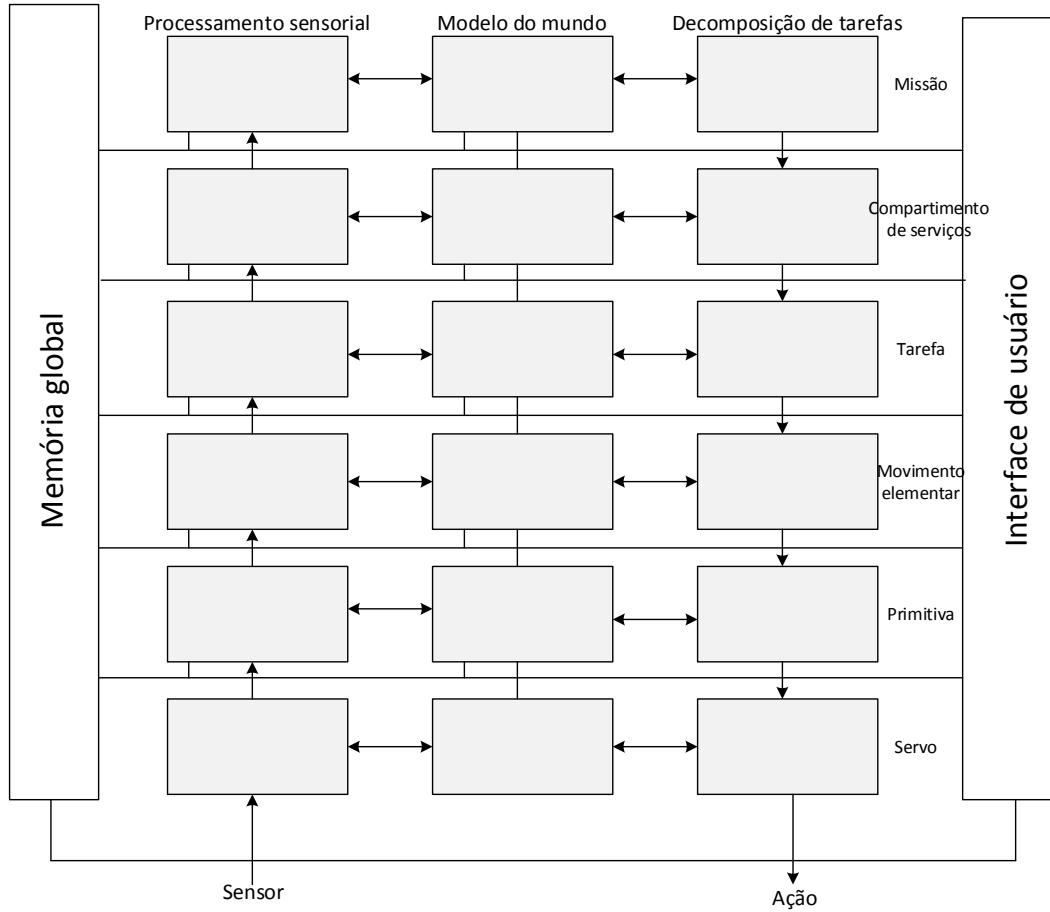


Figura 2.6: Arquitetura NASREM.

Intelligent Mobile Robot System

Em 1991, Saridis [17] cria o *Intelligent Mobile Robot System* (IMRS) baseado na teoria de inteligência hierárquica de controle [18]. Saridis utiliza redes de Petri como módulos básicos da arquitetura para traduzir os comandos gerados pelo nível de organização em algo comprehensível para o nível de execução.

O IMRS possui a seguinte arquitetura (figura 2.7):

- Nível organizacional (organizador de tarefas): gera tarefas de movimentação de alto nível.

- Nível de coordenação: funciona como uma interface entre o nível organizacional e o de execução. O nível é composto por um remetente e alguns coordenadores. O remetente recebe o plano da tarefa do organizador, decompõe a tarefa em ações de controle e remete aos coordenadores. Os coordenadores traduzem os comandos de controle em instruções de operação e transmite ao nível de execução.
- Nível de execução: executa a instrução proveniente do nível de coordenação e reporta seus resultados a ele.

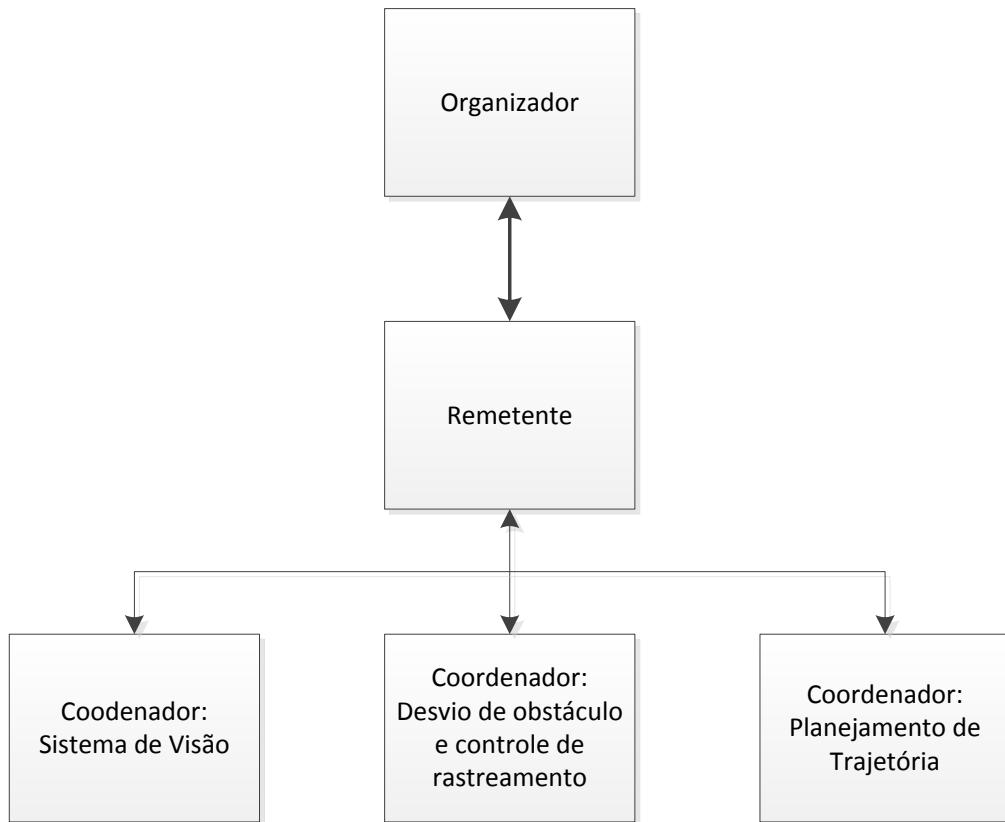


Figura 2.7: Arquitetura de Saridis

A modelagem do sistema utilizando redes de Petri proporcionou algumas funcionalidades essenciais em uma arquitetura robótica, destacam-se: a capacidade de executar duas tarefas simultaneamente (por exemplo, movimentação e planejamento

de trajetórias); e o *Input Semaphore*, que impede um processo de ser executado até outro ser finalizado.

Saridis salienta os benefícios das PNT:

- Redes de Petri podem ser usadas como módulos básicos para sistemas de controle de missão de robôs móveis.
- A comunicação e conexão de módulos são eficientes entre redes de Petri.
- Controle e mecanismo de comunicação para coordenação de tarefas de um robô móvel podem ser realizados com redes de Petri.

A arquitetura de Saridis é uma contribuição importante por criar um nível organizacional, separando o nível do desenvolvedor de baixo nível e um nível de alto nível para um operador (usuário). Além disso, as redes de Petri assumem um importante papel como módulo básico de controle para seu sistema IMRS. As redes de Petri foram originalmente introduzidas para descrever as comunicações de máquinas de estados finitos (FSMs), possibilitando flexibilidade e robustez. É provado que redes de Petri são uma excelente ferramenta para modelagem de sistemas, sobretudo quando envolvem tarefas conflitantes ou simultâneas [19].

MARIUS

Em 1996, Silva et al. [20] (Institute for Systems and Robotics, Lisboa) projetaram, desenvolveram e testaram um sistema de controle de missão para o MARIUS, robô autônomo submarino.

A arquitetura do MARIUS é descrita abaixo, figura 2.8:

- *Vehicle Support System* (VSS) - Controla a distribuição de energia aos hardwares instalados no veículo, monitora consumo de energia e detecta falhas de hardware, podendo enviar comandos de emergência.
- *Actuator Control System* (ACS) - Controla a velocidade de rotação dos propulsores e posição dos ailerions e lemes. Os *Set Points* dos atuadores são dados pelo *Vehicle Guidance and Control System* (VGCS) e os dados dos atuadores são transmitidos para o *Mission Control System*.

- *Navigation System* (NS) - Estima posição linear e velocidade do veículo, orientação e velocidade angular. O sistema funde informações do *Positioning System (Long Baseline unit)* e *Motion Sensor Integration System*, o qual inclui diversos sensores. As saídas do NS são entradas do VGCS, e enviadas ao MCS.
- *Vehicle Guidance and Control System* (VGCS) - Recebe como entrada as trajetórias de referência pelo MCS, e os dados de navegação do NS. Suas saídas são *Set Points* para velocidade de rotação e outros atuadores do ACS, tal que o veículo siga a trajetória desejada mesmo com incertezas e distúrbios.
- *Communication System* (COMS) - Controla o link bidirecional usado pelo operador para passar missões ao MCS, e pelo veículo para passar status de missão ou estados do veículo.
- *Environmental Inspection System* (EIS) - Coleta dados do ambiente com diversos sensores (inclusive câmeras), como temperatura, pressão, pH. É controlado pelo MCS.
- *Data Logging System* (DLS) - Adquire e armazena dados do veículo.
- *Mission Control System* (MCS) - Sequencia e sincroniza a execução das tarefas básicas do veículo para uma determinada missão e provê a recuperação em caso de falhas.

Silva desenvolveu os softwares CORAL e ATOL para implementação das VPs e STs em redes de Petri de forma que um usuário final, como um operador, pudesse facilmente criar seus MPs. A grande contribuição

2.2.3 Análise crítica

A abordagem deliberativa simula, de certa forma, o processo de planejamento e tomada de decisão do ser humano. Há um núcleo (cérebro) que processa todos os dados sensoriais e armazena o mundo, isto é, o ambiente em que o robô está inserido, de maneira simbólica, geométrica ou outros tipos de mapeamento. Além disso, o núcleo planeja todas as ações para uma determinada tarefa, consultando

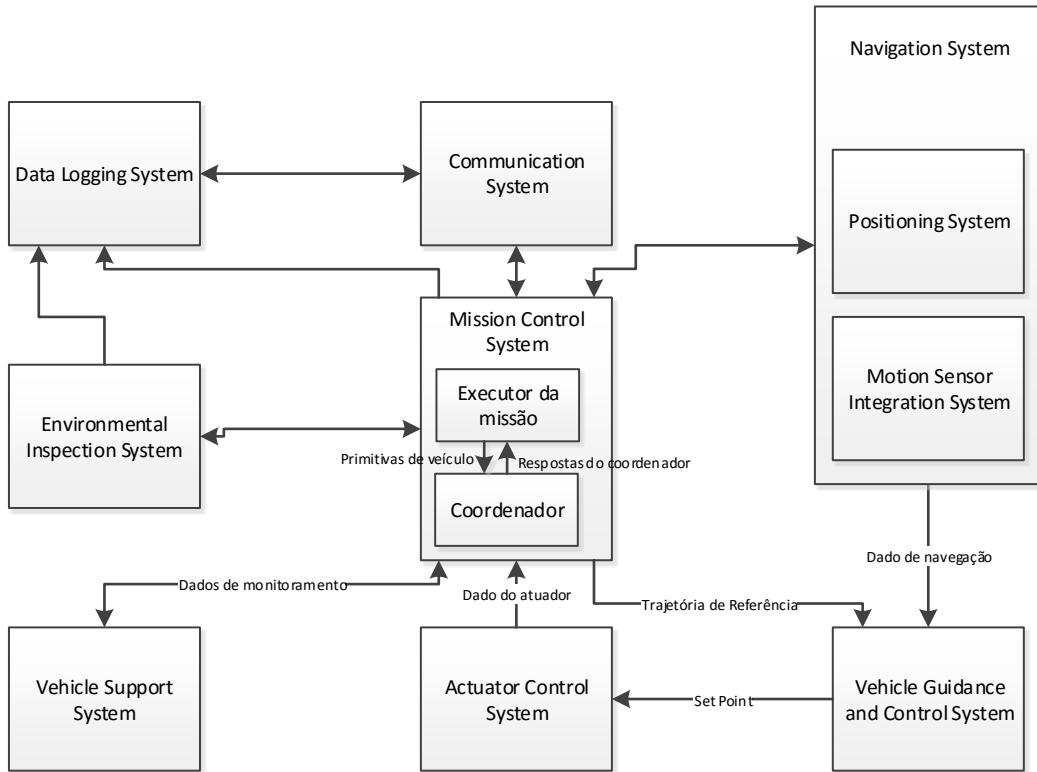


Figura 2.8: Arquitetura do Veículo MARIUS

sua ideia de mundo intensivamente. Há, também, sensores que enviam suas novas informações periodicamente para o núcleo, (órgãos receptivos: visão, olfato, e etc), atualizando o mundo. E há atuadores (músculos) necessários para a realização das tarefas (figura 2.9).

É fácil observar que a arquitetura deliberativa é dependente do modelo de mundo armazenado e suas atualizações periódicas. Portanto, a utilização da abordagem deliberativa em ambientes extremamente dinâmicos pode ser muito custosa devido às atualizações e ao replanejamento. Além disso, a arquitetura SPA dificulta a criação de sistemas em tempo real eficientes. Dessa forma, robôs móveis em ambientes muito dinâmicos, como o carro autônomo da Google (figura 2.10), não são aplicações favoráveis para esta arquitetura.

O controle de missão é presente em arquiteturas deliberativas, mas de maneira primitiva, normalmente possuindo apenas as funcionalidades de: quebrar as missões em tarefas menores que o robô possa executar, gerenciar as tarefas, e monitoramento

interno. Entretanto, não desenvolve uma interface com o usuário, a possibilidade de alteração de tarefas em tempo de execução, e feedback ao operador em tempo real.



Figura 2.9: Comparativo da arquitetura deliberativa com o ser humano.



Figura 2.10: Projeto da empresa Google 2011 - para o desenvolvimento de um carro autônomo.

2.3 Paradigma reativo

Sistemas de arquitetura reativa também são chamados de sistemas baseados em comportamentos. Os robôs são programados para agir através de ativação de uma coleção de comportamentos primitivos de baixo nível. De acordo com [21], as principais características de sistemas puramente reativos são:

- Comportamentos são como elementos construtivos: são um par sensor-motor, onde o sensor provê informação necessária para o motor executar uma ação

reativa, como desvio de obstáculo, atrair-se a objetivos, escapar de predadores e etc.

- Não há criação ou manutenção precisa do modelo do mundo. Os sistemas reagem ao estímulo do mundo, extremamente útil para mundos dinâmicos e hostis.
- Comportamentos de animais são normalmente utilizados para modelar esses sistemas.

Controle reativo é uma técnica que une percepção e ação, tipicamente no contexto de comportamentos motores, para produzir respostas robóticas, em tempo real, em mundos dinâmicos.

Em 1986, um dos primeiros estudos em sistemas reativos foi desenvolvido por Rodney Brooks [8]. Este estudo é a base para diversos trabalhos atuais que envolvem robôs reativos móveis. Os desafios de robôs autônomos apontados por Brooks e que ainda ilustram os problemas da atualidade são: *múltiplos objetivos, múltiplos sensores, robustez e extensibilidade*. De acordo com Brooks, esses desafios não são suportados pela arquitetura tradicional (paradigma hierárquico).

Os múltiplos objetivos de robôs móveis podem:

- Ser conflitantes: por exemplo, um robô pode estar tentando alcançar um determinado ponto no espaço, porém evitando obstáculos locais.
- Ter relações de prioridade: por exemplo, um robô que inspeciona trilhos de trêm deve sair dos trilhos ao ouvir o sinal de um trêm chegando, mesmo se estiver finalizando a operação.
- Ser dependentes: objetivos de *alto nível* englobam diversos objetivos de *baixo nível*. No caso do exemplo acima, o robô que sai do trilho para evitar o trêm deve se manter equilibrado para não cair. Artigos recentes, como em [10] separam esses objetivos em *tarefas* (objetivos de *alto nível*) e *primitivas do veículo*.

Robôs são normalmente providos de múltiplos sensores e suas diversas informações podem ser redundantes, conflitantes ou complementares, podendo ser

utilizadas para uma mesma tarefa do robô. Por exemplo, encoders para odometria e câmeras fixas ao robô podem ser utilizados para localização, de forma que se complementem. Os sensores podem apresentar erros ou resultados conflitantes, portanto a fusão da informação de múltiplos sensores, a determinação de seus graus de confiabilidade e em quais tarefas devem ser considerados são decisões que o robô deve saber fazer.

Um robô deve ser robusto, isto é, em caso de falha de um sensor, o robô deve se adaptar e utilizar os outros sensores que ainda funcionam para realizar as tarefas. Ou em caso de alterações no ambiente, o robô deve ser capaz de cumprir determinadas funções essenciais.

A extensibilidade constitui em acrescentar mais sensores e, portanto, aumentar a capacidade do robô, sendo possível a execução de novas tarefas. Porém, esta afirmação é normalmente criticada por alguns autores já que existe um limite imposto pelo processamento do robô, já que um novo hardware (sensor) é adicionado, mas o processamento (computador) não é substituído e sua capacidade não é aumentada.

Brooks propõe um dos primeiros sistemas baseados em comportamentos, uma arquitetura que tem como objetivo descentralizar a tomada de decisão de um modelo central, como pode ser visto na figura 2.11. O autor comenta que essa decomposição conduz a uma arquitetura radicalmente diferente para robôs móveis, em estratégias de implementação a nível de hardware e com grandes vantagens em robustez, desenvolvimento e teste.

2.3.1 Robôs reativos

Antes mesmo de Brooks, ou seja, antes da formalização de toda a teoria em sistemas reativos, simples robôs eram criados na lógica de controle reativo e de comportamentos. Em 1953, por exemplo, Grey Walter [22] desenvolveu uma “tartaruga” elétrica capaz de se movimentar pelo ambiente, evitando luz intensa (“ameaças”) e atraída por certos objetivos. Vale observar a característica de perceber baixo nível de bateria e procurar uma estação de recarga, comportamento que se sobrepõe aos outros. Os comportamentos são simples: dirigir-se para luz fraca; fugir de luz forte; e evitar obstáculos. Não há representação abstrata do mundo (figura 2.12).

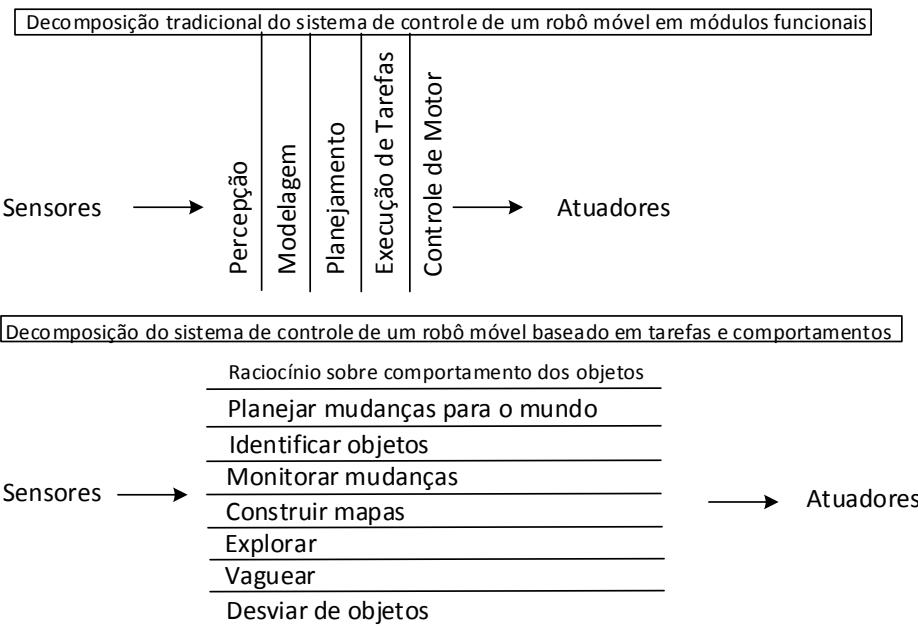


Figura 2.11: Arquitetura para sistema de controle de robôs móveis por Brooks

Em 1984, veículos simples e puramente reativos com os pares clássicos sensor-motor foram desenvolvidos pelo psicólogo Braitenberg [23], a fim de simular sentimentos, como covardia, agressividade e outros (figura 2.13).

Em 2002 até os dias atuais, o robô Roomba da empresa iRobot ganha destaque comercial e executa uma simples tarefa doméstica: limpar o chão. Em sua arquitetura, o robô Roomba possui apenas algumas funções reativas, como esquivar-se e locomover-se, e não possui o modelo do mundo [24] (figura 2.14).

2.3.2 Arquiteturas reativas

As arquiteturas reativas foram formalmente introduzidas em 1986 por Rodney Brooks. O roboticista, futuro fundador da empresa iRobot, vivenciou uma época de processadores lentos e de alto custo, dificultando o uso de diversos sensores e a armazenagem do modelo do mundo no robô. Simulações de tarefas, atualizações do mundo e análise de sensores como câmeras eram extremamente complexas para robôs e impossíveis de serem executadas em tempo real. Brooks vislumbrou como solução o processamento paralelo, evitar o uso de um modelo do mundo e criar

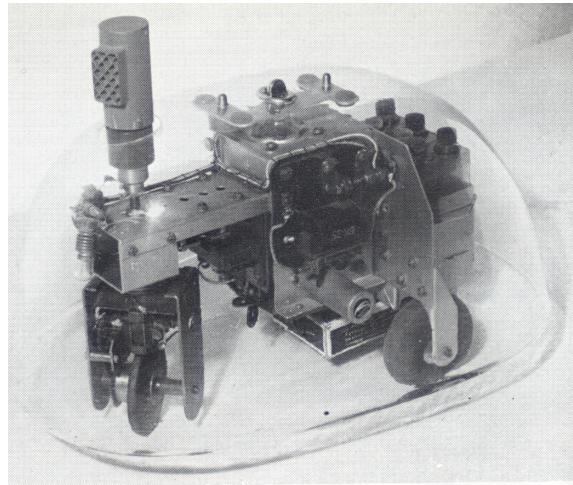


Figura 2.12: A tartaruga de Grey Walter.

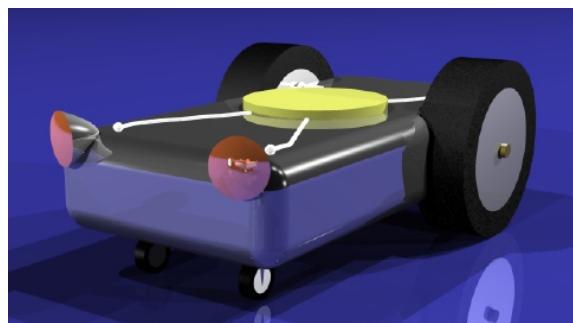


Figura 2.13: Veículos de Braitenberg.



Figura 2.14: Robô comercial Roomba da iRobot.

módulos puramente reativos, pares sensor-motor, que juntos compõem o robô.

Arquiteturas reativas utilizam informações locais do meio, obtidas pelos sensores do robô. Estas são simplificadamente tratadas, de forma que a ação ao estímulo é tomada rapidamente e, assim, os robôs reativos podem responder de forma mais

rápida às variações do ambiente. A arquitetura define como a informação é mapeada em uma ação e como é feita a coordenação dos pares estímulo-ação (comportamentos reativos).

De acordo com Arkin [5], há duas classes predominantes para a função de coordenação: competitiva e cooperativa.

Um conflito ocorre quando dois ou mais comportamentos estão ativos ao mesmo tempo e possuem respostas diferentes. Nesse caso, a coordenação de forma competitiva entra em ação, escolhendo um dos comportamentos para a ação final do robô. A prioridade de um comportamento sobre os demais pode ser definida explicitamente em uma hierarquia entre os comportamentos, ou pode haver uma votação por uma ação, ou outros métodos. Um exemplo de sistema reativo com coordenação do tipo competitiva é o desenvolvido por Brooks, a arquitetura de subsunção.

Na função cooperativa, a ação do robô é a fusão das respostas de todos os comportamentos ativos. A arquitetura reativa esquema motor de Arkin é um exemplo claro deste tipo de coordenação, onde cada comportamento influencia o movimento do robô por meio de um vetor de força artificial e a ação resultante é determinada pela soma vetorial de todos os vetores de força.

Arquitetura de subsunção

A arquitetura de subsunção é, neste trabalho, destacada e exemplificada, já que sua lógica será essencial para a implementação da arquitetura da DORIS.

Na figura 2.11, Brooks define *Níveis de competência*, que são classes de comportamentos desejados para o robô sobre todos os ambientes que ele pode encontrar. As classes definidas por Brooks são:

0. Evitar contato com objetos (estacionários ou móveis);
1. Vaguear sem rumo e sem bater em objetos;
2. Explorar o ambiente utilizando sensores, definir lugares alcançáveis, e seguir rumo em suas direções;
3. Construir um mini-mapa do ambiente e planejar trajetórias de um lugar para outro;

4. Observar mudanças no ambiente;
5. Raciocinar sobre o ambiente em termos de objetos identificáveis e realizar tarefas relacionadas a certos objetos;
6. Formular e executar planos que envolvam mudar o estado do ambiente como desejado;
7. Raciocinar sobre o comportamento de objetos no ambiente e modificar planos quando necessário;

Cada nível de competência inclui, como subconjunto, os níveis de competência anteriores.

Após a decomposição na nova arquitetura, Brooks define as *Camadas de Controle*, correspondentes a cada nível de competência. A ideia dessa abordagem é adicionar camadas de controle a níveis de competências superiores sem precisar alterar a camada do nível inferior. Inicia-se, portanto, com a camada de controle para o nível zero de competência, esta será testada e não mais alterada. Após, é criada a camada de nível 1, capaz de examinar os dados da camada de nível 0 e injetar dados nas interfaces internas deste nível, suprimindo seu trânsito de dados, figura 2.15.

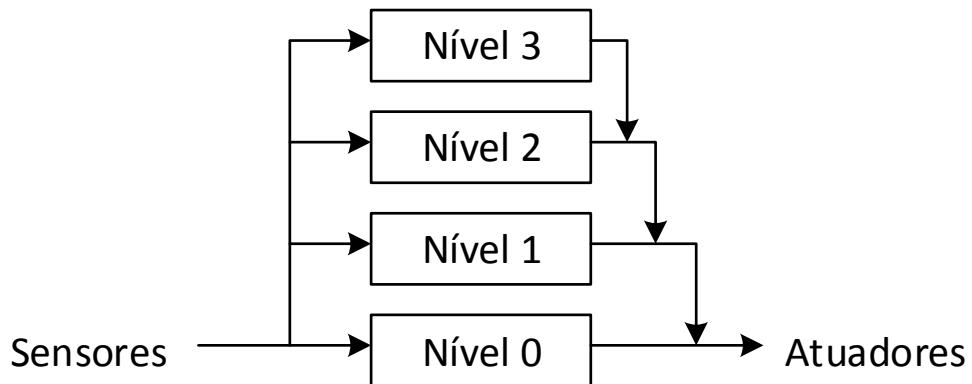


Figura 2.15: Camadas de controle de Brooks

A camada de controle nível zero deve garantir que o robô não entre em contato com outros objetos, estacionários ou móveis. Portanto, o robô deve desviar de

objetos que se aproximam ou parar se houver um objeto fixo em sua trajetória. A camada de controle nível 1, combinada a camada de controle nível 0, permite que o robô vagueie sem colisões. A figura 2.16 mostra o sistema de controle aumentado pelo nível da camada 1.

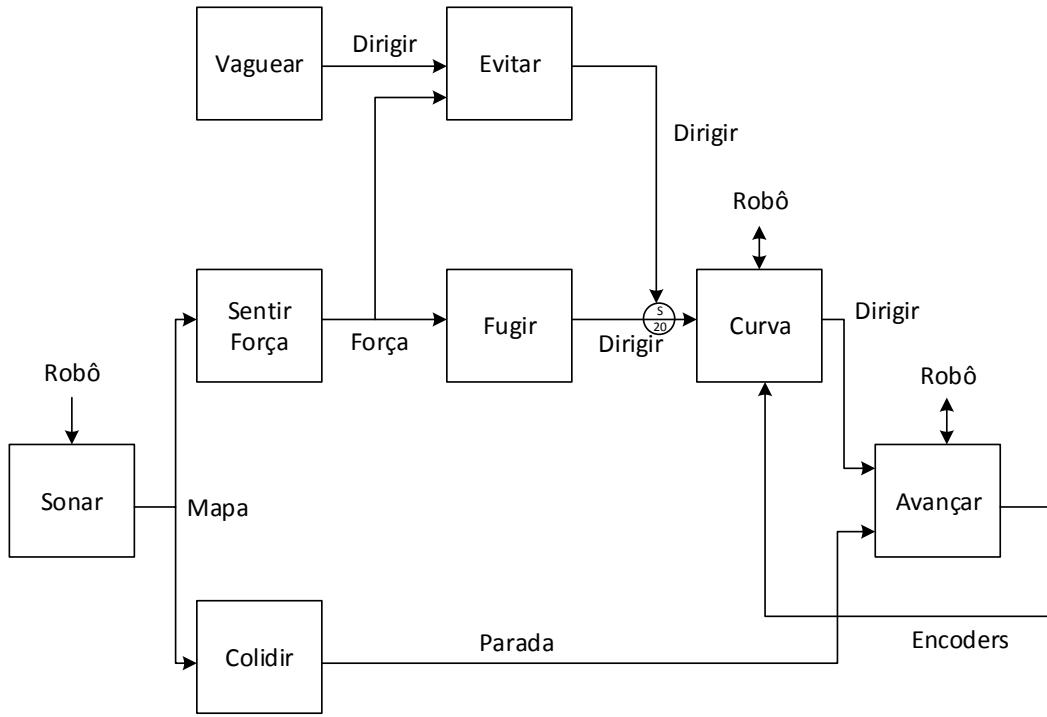


Figura 2.16: Nível 0 e 1 de controle do sistema

A estrutura das camadas de controle foram construídas por um conjunto de pequenos processadores que enviam mensagens uns para os outros. Cada processador é uma máquina de estado finito. A nova arquitetura e essa nova estrutura de camadas com eventos discretos foram a base de diversos sistemas de controle de missão da atualidade.

A nova arquitetura de Brooks é robusta, permite interações dinâmicas, é flexível para integrar novas funcionalidades, em camadas superiores, e fácil para implementar e debugar. Brooks ainda associa sistemas de eventos discretos no controle de robôs autônomos, utilizando como módulos básicos máquinas de estados finitos (*FSM - Finite State Machine*). Como a conexão de diversas *FSM's* não é uma *FSM*, a solução encontrada por Brooks foi acrescentar inibidores e supressores em suas *FSM*,

chamando-as AFSM (*Augmented Finite State Machine* ou máquina de estado finito aumentada, figura 2.17). Porém, o modelo hierárquico dos níveis cria uma certa inflexibilidade nos níveis inferiores.

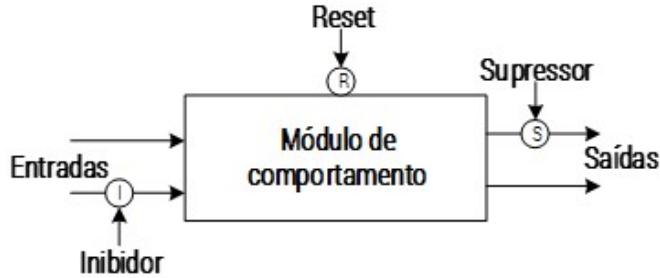


Figura 2.17: Módulo básico da arquitetura de subsunção (AFSM).

Apesar dos pontos positivos, a arquitetura apresenta problemas com escala e contextualização (*situatedness*).

O problema com escala é resultado das interconexões, que podem crescer de maneira fatorial em relação ao número de comportamentos. A contextualização é um problema de sistemas de subsunção, onde subsistemas são incluídos em um sistema mais amplo. Todos os comportamentos estão operando ao mesmo tempo e anulando as saídas de outros comportamentos, de forma que apenas uma é a saída do veículo. Portanto, há computação ineficiente.

Apenas em 1996, Bellingham e Consi [25] propuseram um controle por camadas (*Layered Control*) a fim de resolver o problema de escala. Há um módulo de processamento de sensores que disponibiliza os dados aos comportamentos, resolvendo o problema de interconexões, já que antes as camadas superiores deveriam se conectar às inferiores para obter os dados dos sensores. Porém, Bellingham não resolveu o problema utilizando uma arquitetura do tipo subsunção: as camadas são associadas com um número de prioridade, de forma que saídas conflitantes são resolvidas com esse número. Esta solução pode gerar uma certa inflexibilidade, visto que a adição de um novo comportamento pode alterar toda a estrutura de prioridades previamente estabelecida.

A solução para o problema de contextualização só foi resolvida em 2000 por Bennet [26]. No *State Configured Layered Control* (SCLC), múltiplos conjuntos de comportamentos simples são escolhidos de uma biblioteca de comportamentos

e são executados em cada fase da missão. A vantagem é o número reduzido de comportamentos executados ao mesmo tempo. Esta é uma estratégia comum em arquiteturas de controle híbrido.

O AUV Eric Desenvolvido pelo Key Center Robotics Laboratory, University of Technology, Sydney [27], o AUV Eric segue a arquitetura de subsunção desenvolvida por Brooks com alguns avanços. Foram desenvolvidos três níveis de competência, que descrevem a capacidade do sistema: nível de preservação, nível de exploração e nível de socialização. O nível de preservação realiza o desvio de obstáculos, a exploração executa navegação de alto nível, e a socialização habilita o comportamento de seguir objetos.

Arquitetura Esquema Motor

Em 1987, Arkin [28] utiliza a teoria de esquemas (psicologia) proposta por [29] para desenvolver sua função de coordenação cooperativa para sistemas reativos, chamado de Esquema Motor. Neste sistema, as respostas dos comportamentos aos estímulos são representadas por vetores (magnitude e orientação) e a coordenação é alcançada pela adição dos vetores, produzindo um vetor resultante. Não há hierarquia pré-definida entre comportamentos, todos os comportamentos ativos contribuem para a saída do sistema com sua resposta individual (vetor) e ganho associado. O ganho do vetor (ou peso) é um parâmetro para dar flexibilidade, possibilidade de aprendizado e adaptação do robô caso este não seja fixo.

Em uma tarefa de navegação, é fácil imaginar como funciona o método do Esquema Motor. São definidos alguns esquemas motor (comportamentos), como mover-se em direção ao objetivo, evitar obstáculos, desviar, escapar, e outros, e cada comportamento responde com um vetor que representa velocidade (magnitude) e direção (orientação) que o robô deve seguir. A soma dos vetores resulta na direção e velocidade final do robô. Ao perceber um obstáculo, por exemplo, os vetores do comportamento *desvio de obstáculos* possuem magnitude superior aos outros e orientação para fora do obstáculo (vetor de repulsão), de forma que o robô executa o desvio, em vez de colidir.

Assim como a arquitetura de subsunção, no Esquema Motor, os esquemas agem

de maneira distribuída, paralela e são modulares. O sistema apresenta vantagem em relação à arquitetura de camadas por sua dinâmica, já que os esquemas podem ser instanciados e desinstanciados a qualquer momento, e fácil reconfiguração. Apesar do importante e atraente resultado em tarefas de navegação, os esquemas motores dominaram apenas esse nicho da robótica e, mesmo neste nicho, outras tarefas normalmente não são executadas com esta arquitetura.

2.3.3 Análise crítica

Se a abordagem deliberativa simulava o processo de planejamento e tomada de decisão do ser humano, os sistemas de arquitetura reativa simulam outra importante função da medula espinhal, pertencente ao nosso sistema nervoso central: o circuito reflexivo, ou arco reflexo. O reflexo é uma resposta involuntária rápida, consciente ou não, originado de um estímulo externo e realizado antes mesmo de o cérebro tomar conhecimento do estímulo periférico.

Dessa forma, não há planejamento, não há modelo de mundo, apenas uma reação ao estímulo. Esse comportamento é extremamente importante e necessário para a sobrevivência do ser humano. Por exemplo, quando encostamos a mão em uma panela quente, temos a reação imediata de retirar a mão sem intervenção do cérebro, sem replanejamento, cujo processamento levaria tempo suficiente para causar danos severos. É fácil, portanto, perceber que tais comportamentos são necessários também a robôs.

Robôs simples, como por exemplo um robô para limpar trilhos de trêm, ou robôs para limpar o chão, podem ter o custo reduzido e executar extraordinariamente bem suas tarefas sem a necessidade de planejamento e modelos complexos do mundo. Durante a execução de sua tarefa, um robô limpador de trilhos só precisa saber que é necessário sair do trilho ao avistar um trêm, logo basta um par estímulo-motor e uma supressão de tarefa (no caso, a tarefa de continuar limpando o trilho). Os grandes motivadores do aparecimento da arquitetura reativa foram as aplicações simples para alguns robôs e a baixa capacidade dos processadores da época.

Não é comum, na literatura, encontrarmos controle de missão em arquiteturas reativas e é comum alguns autores se referirem a esses robôs como *non-taskable systems*, isto é, sistemas sem atribuição de tarefas. Não há interface com o usuário,

mas sim com o programador, que deve alterar comportamentos a fim de gerar uma nova tarefa ou aplicação.

A tabela 2.1 mostra a comparação das arquiteturas analisadas até agora.

Tabela 2.1: Tabela comparativa de arquitetura deliberativa e reativa

Arquiteturas deliberativas	Arquiteturas reativas
Modelo interno completo e preciso	Informações sensoriais locais
Planejamento	Ações pré-definidas às informações sensoriais
Maior flexibilidade na definição de tarefas e objetivos	Sistemas mais dedicados às tarefas e problemas específicos
Resposta lenta às mudanças no ambiente	Resposta rápida às mudanças do ambiente

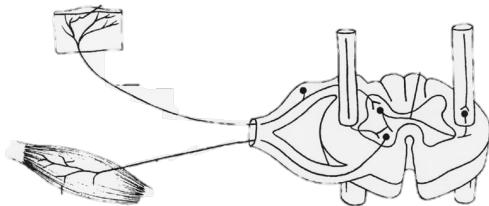


Figura 2.18: Analogia de sistemas reativos com o ser humano.

2.4 Paradigma híbrido ou deliberativo/reactivo

Arquiteturas deliberativas e reativas apresentam vantagens e desvantagens e que muitas vezes se opõe. Por exemplo, é muito ineficiente utilizar uma arquitetura deliberativa em um ambiente extremamente dinâmico, assim como é ineficiente utilizar arquitetura reativa em uma linha de montagem (manipuladores industriais). É natural pensar que integrando, de alguma forma, as duas arquiteturas, formando uma arquitetura híbrida, será possível absorver as vantagens de ambas.

Atualmente, os robôs com arquitetura híbrida predominam. Em muitas aplicações de robôs móveis mais complexas, fica claro que formas de conhecimento

mento do mundo na arquitetura robótica permitem que a navegação dos robôs seja mais flexível, eficiente e geral. A arquitetura híbrida tenta combinar os métodos simbólicos da IA e seu uso de representação abstrata do modelo do mundo, mas mantém o objetivo de prover robustez, resposta em tempo real e flexibilidade dos sistemas puramente reativos. Arquiteturas híbridas podem permitir a reconfiguração de controles reativos baseado no conhecimento do mundo através da sua capacidade de raciocinar sobre os componentes comportamentais subjacentes.

O principal problema no paradigma híbrido é como desenvolver uma metodologia unificada de arquiteturas que garanta um sistema capaz de executar planos de uma maneira robusta, como a arquitetura reativa, e, ao mesmo tempo, ter um entendimento de alto nível da natureza do mundo e um modelo da intenção do usuário.

2.4.1 Robôs híbridos

Entre 1986 e 1987 [30], Arkin foi o primeiro a criar uma arquitetura híbrida e implementar em um robô. O objetivo do robô era reconhecer o ambiente de trabalho, planejar uma trajetória e navegar até o ponto desejado, evitando obstáculos estáteis e móveis (figura 2.19).

Em 1996, o AUV com arquitetura híbrida Phoenix, desenvolvido em Naval Post-graduated School, Monterey, foi testado em Moss Landing California [31], figura 2.20.

Desde 2003 com o Spirit até os dias atuais com Opportunity e Curiosity (figura 2.21), a NASA vem desenvolvendo robôs de alta tecnologia, com grande robustez de hardware e software.

2.4.2 Arquitetura híbrida

Em [5], Arkin aponta quatro possíveis estratégias para o projeto de arquiteturas híbridas:

- Seleção: a deliberação é vista como um configurador. O planejador determina a composição de comportamentos e parâmetros usados durante a execução. O planejador pode reconfigurá-los quando necessário devido às falhas no sistema.

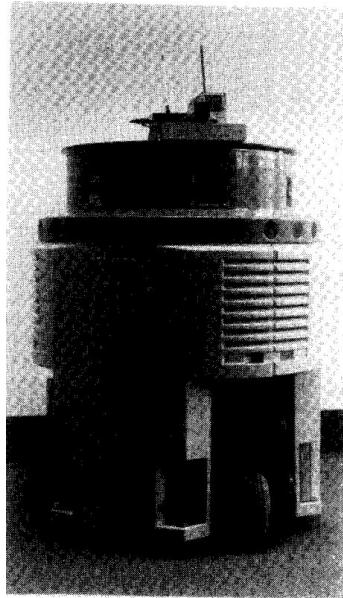


Figura 2.19: Robô George.



Figura 2.20: O AUV Phoenix.



Figura 2.21: Robô Curiosity, o robô da NASA explorador do planeta Marte.

- Conselho: a deliberação é vista como um aconselhador. O planejador sugere mudanças que o controle reativo pode ou não usar.
- Adaptação: a deliberação é vista como um adaptador. O planejador continuamente altera os componentes reativos ativos de acordo com as mudanças nas condições do mundo e requisitos de tarefas.

- Adiamento: a deliberação é vista como um recurso para ser usado em último caso. Neste caso, os planos só são elaborados quando necessários.

Em [1], Murphy afirma que apesar de arquiteturas híbridas variarem em como a função deliberativa (o planejador) atua no sistema, os componentes implementados em uma arquitetura híbrida são semelhantes, possuindo geralmente os seguintes módulos:

- Sequenciador: agente que gera o conjunto de comportamentos a serem usados para completar uma subtarefa, e determina qualquer sequência e condições de ativação. A sequência é normalmente representada como uma rede de dependências ou FSMs, mas o sequenciador gera essa estrutura ou dinamicamente a adapta.
- Gerenciador de recursos: agente que aloca recursos aos comportamentos. Este agente está diretamente ligado aos sensores do robô, podendo verificar qual o sensor é mais adequado para cada situação do robô.
- Cartógrafo: agente responsável por criar, armazenar e atualizar o mapa, os modelos do mundo.
- Planejador de missão ou controle de missão: agente que interage com o usuário, traduz a mensagem do usuário para os termos do robô, e constrói um plano de missão. Por exemplo, um robô “assistente” poderia receber o comando “Robô, traga um xícara com café”. O planejador de missão interpreta o comando e subdivide nas missões: procurar uma xícara, servir café (supondo que este já esteja pronto) e, levá-la ao usuário.
- Monitor de desempenho e solucionador de problemas: é o agente que permite ao robô perceber se está progredindo no cumprimento de sua tarefa.

Neste trabalho, são destacadas as arquitetura híbridas: Autonomous Robot Architecture (AuRA), AUV Phoenix, CLARAty e Stanley.

AuRA

A Autonomous Robot Architecture (AuRA) foi a primeira arquitetura híbrida, desenvolvida em 1986 por Arkin [28]. De acordo com Murphy [1], AuRA pertence

ao estilo de arquitetura híbrida *administradora* (*Managerial*), reconhecida por sua decomposição similar a um gerenciamento de negócios. Há agentes superiores que realizam planejamento em alto nível e passam o plano a subordinados, os quais refinam os planos e coletam os recursos, e, então, estes são passados ao último nível de agentes, os comportamentos reativos. De acordo com Arkin, AuRA é um projeto de arquitetura com estratégia de Seleção.

AuRA é composto por cinco subsistemas, dos quais dois pertencem à porção deliberativa: Planejador e Cartógrafo, o terceiro é um subsistema de sensores, quarto é subsistema de gerenciador de comportamentos (gerenciador esquema motor) e o último subsistema de controle homeostático.

O planejador é responsável pela missão e o planejamento de tarefas. É subdividido em três componentes: planejador da missão, navegador e piloto, e seus módulos são executados sequencialmente, tornando-se mais específicos e detalhados. A lógica é hierárquica: o planejador de missão envia trechos da missão ao navegador, o qual envia trechos de trajetória para o piloto, que determina ações ao controlador de baixo nível. O planejador de missão também funciona como interface ao usuário. A utilização do mapa interno do robô por cada módulo é diferente, enquanto o planejador usa o mapa global, o piloto recebe informações locais. Vale observar que, quando o modelo do mundo é atualizado, muitas vezes não há necessidade de o planejador atualizar toda a missão e recomeçar o ciclo de planejamento, o piloto pode recalcular a trajetória local.

O Cartógrafo encapsula todo o mapa e tem a funcionalidade de receber mapas a priori (operador pode fornecer um mapa inicial). Os três componentes do planejador interagem com o Cartógrafo para obter a trajetória a ser seguida, quebrada em segmentos.

O subsistema do AuRA que gerencia comportamentos utiliza a arquitetura reativa esquema motor. O esquema motor, como visto na subseção 2.3.2, representa cada ação em campos potenciais (vetores) e a resposta do sistema é a soma dos vetores.

O controle homeostático, quinto subsistema, assume um papel intermediário entre os sistemas deliberativo e reativo. Sua função é modificar a relação entre comportamentos, modificando os ganhos dos vetores (subseção 2.3.2). Considere o seguinte

exemplo, para melhor entendimento deste subsistema: um veículo (robô), operando no planeta Marte em um ambiente rochoso, têm como tarefa remover fisicamente amostras de rochas de várias localidades pelo planeta e levá-las a um veículo de retorno, o qual tem uma data fixa de lançamento. O robô é provido com ganhos padrões em seus comportamentos, os quais produzem uma execução conservadora da operação, por exemplo garantindo que o robô esteja sempre 2 metros de distância dos obstáculos em seu caminho. No começo da missão, a execução conservadora parece razoável, mas agora considere próximo à data limite de decolagem do veículo de retorno: se o robô estiver próximo ao veículo de decolagem, ele deveria cortar caminhos, reduzindo sua margem de evitar obstáculos para realizar a entrega, sacrificando sua própria existência pela missão.

AuRA poderia integrar o controle homeostático à parte deliberativa da arquitetura, mas foi motivado pela biologia a colocá-lo em outra área, chamada de subconsciente, a qual, em animais, modifica sempre a área consciente em resposta a necessidades internas. A figura 2.22 mostra os cinco subsistemas da arquitetura AuRA. A tabela 2.2 resume a arquitetura.

Tabela 2.2: Resumo da arquitetura AuRA

Sequenciador	Navegador, Piloto
Gerenciador de recursos	Gerenciador esquema Motor
Cartógrafo	Cartógrafo
Planejador de missão	Planejador de Missão
Monitor de desempenho	Piloto, Navegador, Planejador de missão

Arquitetura de três camadas

De acordo com Murphy, esta arquitetura pertence ao estilo Estado-hierárquico por ser organizado em três camadas baseadas no estado de conhecimento: Planejador, Executivo e Funcional. A arquitetura de três camadas é a mais aplicada em robôs de grande complexidade , como AUVs e robôs do desafio DARPA (GRACE e Stanley). Para melhor entendimento das funções das três camadas, será utilizado o exemplo empregado em [32]: considere um robô de entregas em escritório, operando em um ambiente conhecido (mapeado).

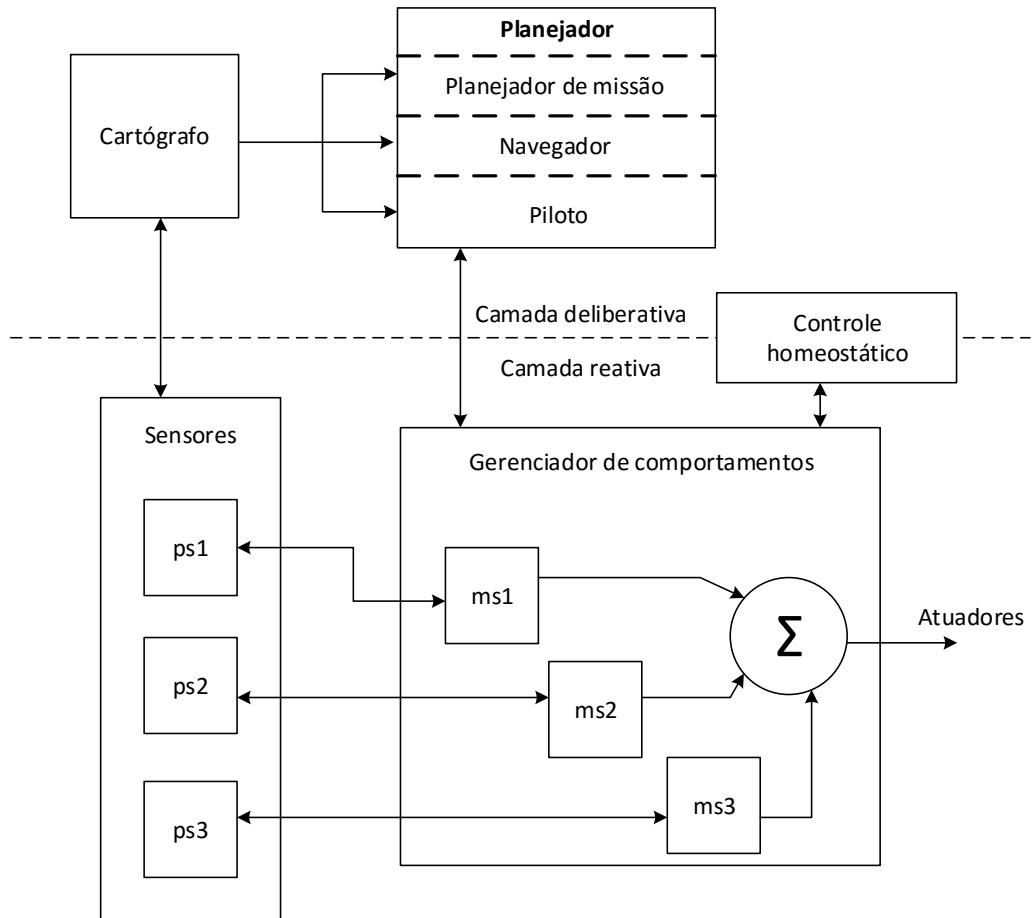


Figura 2.22: Arquitetura AuRA.

O planejador é uma camada composta pelo planejador de missão e o cartógrafo. O planejador de missão gera objetivos e planos estratégicos, e os repassa para a camada intermediária, chamada Executivo. O cartógrafo é responsável por armazenar e atualizar o modelo do mundo, além de poder prover diferentes modelos para as diferentes funções do robô, a fim de otimizar a execução.

No exemplo do robô de escritório, o Planejador analisa as entregas do dia, pode verificar os recursos do robô, atualiza o mapa, pode determinar as melhores rotas de entrega e agendamento, pode estimar quando o robô deve se recarregar, e replaneja quando necessário, por exemplo, quando um escritório estiver fechado, o robô deve alterar o agendamento para realizar a entrega mais tarde.

A camada Executivo utiliza técnicas de planejamento reativo para selecionar um

conjunto de comportamentos de uma biblioteca de comportamentos, e desenvolve uma rede de tarefa (*task network*) especificando a sequência de execução para os comportamentos a um subobjetivo particular. De acordo com Murphy, o Executivo seria a camada responsável por sequenciar e monitorar o desempenho de uma arquitetura híbrida. Ele é responsável por traduzir os planos de alto nível da camada Planejador em comportamentos de baixo nível, chamando estes na hora apropriada. É possível que a camada Executivo aloque e monitore os recursos, substituindo esta função do Planejador.

No exemplo do robô de escritório, uma tarefa de alto nível seria entregar uma carta a um escritório. O executivo deveria decompor esta tarefa em subtarefas, deve usar um planejador de trajetórias geométrico para determinar a sequência de corredores ao qual deve passar e interseções que deve virar, anunciar que a pessoa tem uma carta e monitorar se a pessoa pegou a carta. Se a pessoa não receber a carta depois de um determinado tempo, uma falha deve ser acionada, e uma ação de recuperação deve ser tomada, como anunciar novamente à pessoa ou notificar o Planejador a reagendar a entrega.

Os comportamentos selecionados para uma tarefa pelo Executivo formam a camada Funcional. Esta camada é o nível mais baixo nesta arquitetura, conectando diretamente sensores e atuadores. É neste nível que a teoria de controle tradicional reside, como a implementação de controles PID, filtros de Kalman e etc.

No exemplo do robô de escritório, alguns comportamentos possíveis para o robô são: 1) mover a uma determinada localização, evitando obstáculos; 2) mover por um corredor, evitando obstáculos; 3) achar uma porta; 4) achar maçaneta da porta; 5) agarrar a maçaneta da porta; 6) girar maçaneta da porta; 7) atravessar porta; 8) determinar localização; 9) achar número do escritório; 10) anunciar entrega. Estes comportamentos combinam sensores (visão, distância por infravermelho, e outros) e atuadores (motores das rodas, motores do manipulador e outros).

A tabela 2.3 resume os componentes desta arquitetura híbrida.

A arquitetura do AUV Phoenix foi destacada por ser um AUV e, portanto, ser uma aplicação semelhante ao robô em destaque desta dissertação. São famosas as arquiteturas Atlantis [33], sua evolução CLARAty [34], ambas desenvolvidas pela NASA, e o robô Stanley (Stanford).

Tabela 2.3: Resumo da arquitetura de três camadas

Sequenciador	Executor
Gerenciador de recursos	Executor
Cartógrafo	Planejador
Planejador de missão	Planejador
Monitor de desempenho	Planejador

AUV Phoenix Em 1996, Healey [35], California, aplica técnicas de controle híbrido em seu trabalho de desenvolvimento do AUV Phoenix. Healey propõe uma arquitetura de software híbrida com três níveis organizacionais e hierárquicos. Há um aumento de inteligência entre as camadas, do reflexivo, ao procedural, para o deliberativo figura 2.23:

- **Estratégico (planejador)**: utiliza Prolog como linguagem de controle de missão. Desenvolve os comandos que levam o veículo a executar determinada missão.
- **Tático (executivo)**: funções na linguagem C que faz interface com os predicados de Prolog e retorna variáveis booleanas (*true*, *false*). Este nível funciona de maneira assíncrona e retém os dados da missão, além de se comunicar com o nível de execução.
- **Funcional**: controlador em tempo real do veículo, usando mensagens assíncronas. Opera os atuadores do veículo.

O controle híbrido será responsável tanto pela movimentação do veículo, contínuo e síncrono, quanto pela sequência lógica das fases das missões, eventos discreto com transições assíncronas. A arquitetura incorpora detecção de erros e procedimentos de recuperação. O módulo reativo é desenvolvido pela arquitetura de subsunção.

Stanley Em 2006, Sebastian Thrun [36] ganhou o desafio DARPA, em uma competição de veículos autônomos organizada pelo governo dos Estados Unidos a fim de promover o desenvolvimento na área de direção autônoma.

O robô Stanley foi desenvolvido pela Stanford University. É um veículo (Volkswagen Touareg R5) Fora de estrada (*Off Road*) autônomo de alta velocidade, que ga-

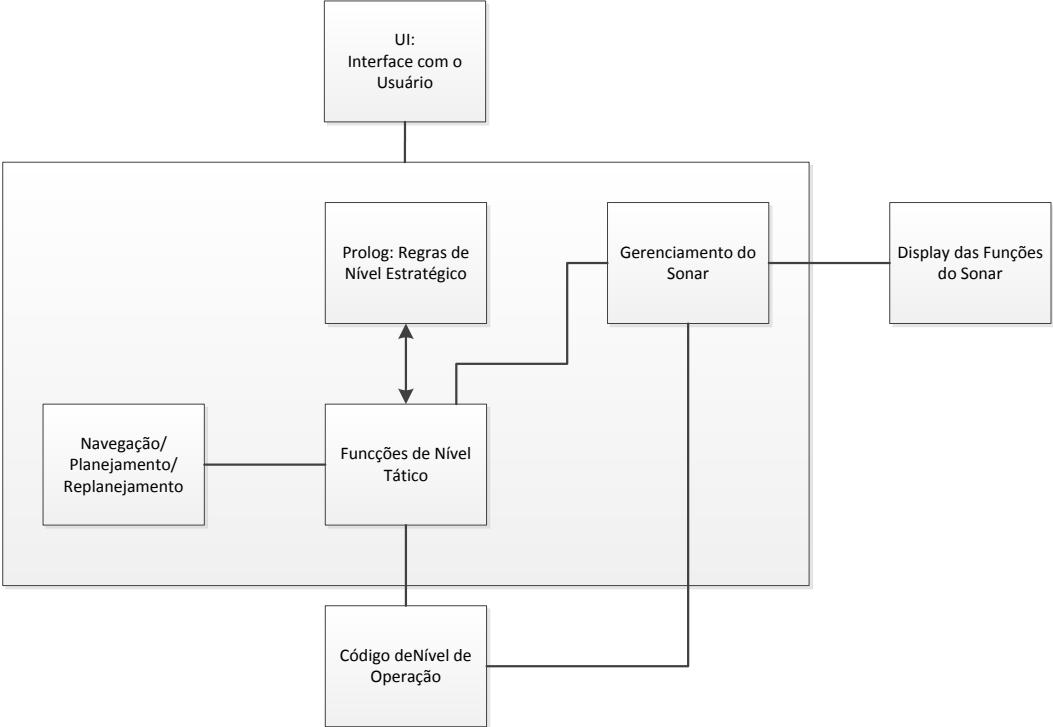


Figura 2.23: Arquitetura do Robô Phoenix de Healey

nhou o desafio de se locomover pela acidentada região do deserto Mojave por mais de 132 milhas sem a interferência de um ser humano. Em grande parte, o desafio DARPA é uma competição de software, pois um motorista equipado com um carro apropriado consegue atravessar o deserto sem muitas dificuldades.

O software desenvolvido é capaz de adquirir dados de sensores, construir modelos do mundo e tomar decisões de direção a uma velocidade de 60 km/h. Neste robô, foi utilizada a arquitetura híbrida de três camadas, como em Atlantis (figura 2.24)

CLARAty Em 2001, Volpe [34] desenvolve uma arquitetura híbrida de duas camadas: Coupled Layer Autonomous Robot Architecture (CLARAty), como uma evolução da arquitetura Atlantis, arquitetura de três camadas. De acordo com Volpe, a arquitetura de três camadas apresenta algumas desvantagens:

- As responsabilidades e tamanho de cada nível é subjetivo ao criador da arquitetura. Portanto, há arquiteturas em que a camada funcional é dominante, outras em que a camada de planejamento domina.

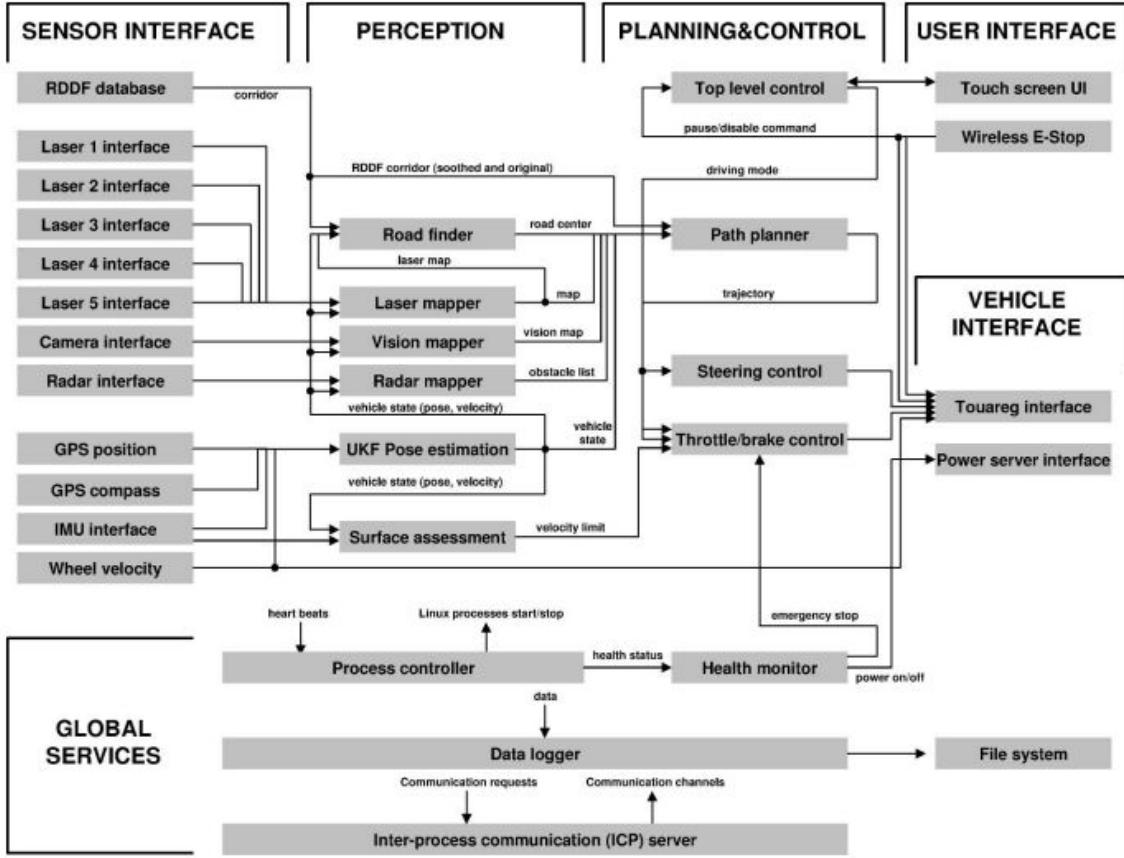


Figura 2.24: Arquitetura do Robô Stanley

- O acesso entre a mais alta camada da hierarquia (planejador) à camada de nível inferior (funcional) é restrita. Apesar de isso ser desejado durante execução, isto separa o planejador de informações da funcionalidade do sistema durante o planejamento. Uma consequência é que planejadores normalmente carregam seus modelos do sistema, que não são derivados (sentido de classe derivada, em programação como C++) diretamente do nível Funcional. Esta repetição de modelos pode gerar certas inconsistências.
- Cada camada pode ter sua própria hierarquia com granularidade variada. A camada funcional é composta por vários subsistemas aninhados, o executivo tem árvores lógicas para coordená-los, e o planejador tem diversas linhas de tempo e horizontes.

De acordo com Volper, a estrutura CLARAty tem duas vantagens principais: representação explícita da granularidade das camadas (em uma representação 3D);

e a mistura das técnicas declarativas e processuais para a tomada de decisões.

A camada funcional é uma interface com todo o sistema de hardware e suas capacidades. É um software orientado a objeto, obtendo assim modularidade de hardware, e estruturação apropriada de software para usar as propriedades de herança, característica extremamente importante para o nível funcional.

2.4.3 Análise crítica

Continuando a analogia de robôs com o funcionamento de animais e, especificamente o ser humano, a arquitetura híbrida visa juntar as duas abordagens anteriores, deliberativa e reativa, e de forma inteligente aproveitar as vantagens de ambas as arquiteturas e aproximar cada vez mais o funcionamento das máquinas com o homem. Nesta arquitetura final, está presente tanto o planejamento exercido pelo cérebro, quanto a camada reflexiva da medula, buscando tornar o robô uma entidade completa (figura 2.25).

A grande dificuldade está em como será realizada essa fusão de arquiteturas, qual será a maneira ótima e que garanta essas vantagens. A abordagem híbrida evolui a cada ano e novos sistemas, como robôs aeroespaciais e carros com direção autônoma, provam que esta arquitetura é eficiente.

A arquitetura híbrida é modular, pois é dividida em camadas, as quais são subdivididas em módulos. O escopo de aplicação é enorme se comparada às outras arquiteturas, como já foi apontado: veículo aeroespacial, carros autônomos, AUV e outras, e como é possível partitionar a arquitetura, pode-se utilizar apenas a área deliberativa ou a reativa e englobar o escopo das duas arquiteturas anteriores. Além disso, a arquitetura híbrida provê robustez, já que apresenta monitor de desempenho e é capaz de adaptação e reconfiguração.

2.5 Robotic Development Environments

A tecnologia em robótica e o número de aplicações de robôs autônomos aumentaram muito nos últimos anos, exigindo o desenvolvimento de softwares robustos, de alta performance, de fácil uso e com possibilidade de simular modelos robóticos, sensores e controle em ambientes virtuais. De acordo com [37], os ambientes de

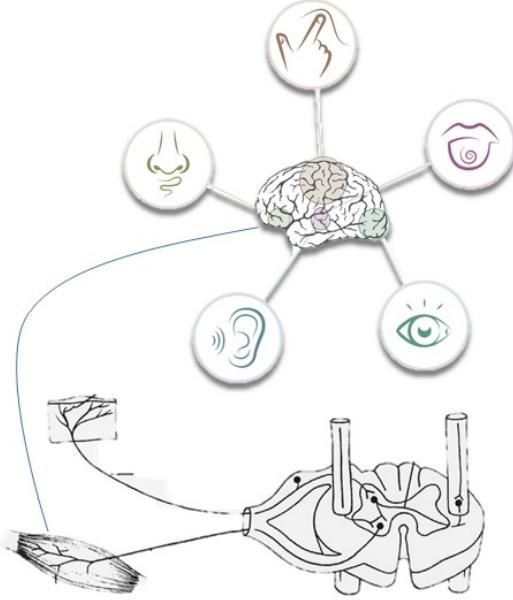


Figura 2.25: Analogia de sistemas reativos com o ser humano.

desenvolvimento em robótica evoluíram devido à emergente interação entre robôs e seres humanos, o que exige grande robustez e simulação prévia, como em aplicações médicas, assistência em ambientes nocivos, e outros. Dessa forma, a comunidade robótica reconheceu a necessidade do desenvolvimento de softwares abertos para simulação e interface em robótica.

As arquiteturas híbridas são, normalmente, compostas em camadas, isto fica explícito, por exemplo, na arquitetura de três camadas. Há linguagens e softwares que visam ajudar na implementação de cada camada. Em seguida, são abordados os softwares e linguagens para cada camada, analisando os principais ambientes de desenvolvimento atuais, destacando vantagens e desvantagens, e justificando a escolha do autor.

2.5.1 Camada Funcional

Os componentes da arquitetura necessitam se comunicar entre si, trocar dados e enviar comandos. A forma de comunicação entre componentes, comumente chamada *middleware*, é o cerne no desenvolvimento de uma RDE, e é uma das mais importantes e restritivas decisões no projeto de uma arquitetura robótica. De acordo com [32], durante o desenvolvimento de arquiteturas, a grande maioria dos problemas e

a maior parte do tempo gasto em depuração (*debugging*) está ligado à comunicação entre os componentes. Há basicamente dois estilos de comunicação: cliente-servidor (*client-server*) e editor-subscritor (*publish-subscriber*).

Em um protocolo de comunicação cliente-servidor, componentes falam diretamente com outros componentes. As vantagens deste protocolo são: definição prévia da interface; e é uma abordagem distribuída para comunicação, já que não há um módulo central que distribui os dados. Uma desvantagem deste protocolo é o *overhead*, que é significativo se muitos componentes precisam de uma mesma informação.

Em um protocolo de comunicação *publish-subscriber* (ou *broadcast*), um componente publica dados e qualquer outro componente pode subscrever (ouvir) a esse dado. Normalmente, há um processo central que roteia os dados entre *publish* e *subscriber*. Em uma arquitetura típica, um componente pode publicar e subscrever a vários tipos de informações. As vantagens desse protocolo de comunicação são: simplicidade e pouco *overhead*; ideal quando não se sabe quantos componentes diferentes necessitarão dos dados (como em muitas interfaces); componentes não ficam sobrecarregados em caso de múltiplos pedidos de um mesmo dado. Suas principais desvantagens são: difícil de *debug*, pois normalmente a sintaxe da mensagem está escondida em uma simples *string* (uma *lista* pode ser enviado como *string* e o componente subscritor retraduz); utilização de um servidor central que distribui as mensagens (recebe dos *publishers* e envia aos *subscribers*), o que pode criar um único ponto de falha e sobrecarga.

A utilização do protocolo de comunicação tipo *publish-subscriber* já aparecia em arquiteturas robóticas antes de Brooks. Em 1983, Elfes [38] idealiza uma arquitetura em módulos e controle distribuído a fim de atingir efetividade em processamento paralelo, flexibilidade de interação com os diversos sensores, distribuir capacidades de decisão e flexibilidade de expansão e modificação do sistema. O sistema de comunicação entre módulos era centralizado e chamado de *Blackboard*.

Em [37] e [39], foram realizadas compilações de RDEs da camada Funcional, apontando suas vantagens, desvantagens, aplicações e os tipos de comunicação. Neste trabalho, são destacados os dois sistemas abertos mais populares: Player e ROS, e, por fim, é justificado a escolha por ROS como o RDE da DORIS.

Player

Em 2001, Player/Stage [40] foi projetado para ser uma interface de programação, não um ambiente de desenvolvimento, focando em programação de dispositivos (drivers) e sensores, em vez de robôs. Em Player, os dispositivos são independentes entre si e são registrados em um servidor, podendo ser acessados por clientes (programas de controle). Cada cliente usa uma conexão com o servidor para transferência de dados, possibilitando operações concorrentes. Múltiplos clientes podem subscrever a um mesmo dispositivo e podem enviar comandos simultaneamente, porém não há fila, logo comandos antigos são descartados. Stage, a segunda parte do software, é um simulador de dispositivo.

O objetivo de Player é a separação de interface e função. O fato de servidores se comunicarem via soquete TCP/IP possibilita que clientes possam ser escritos em qualquer linguagem de programação que possua esse suporte, por exemplo Python, C++, Java e outros. Clientes podem operar em qualquer host que possua conexão, possibilitando independência de localização.

Robot Operating System - ROS

ROS é um software aberto que provê camada de abstração de hardware (CAH - hardware abstraction), controle baixo nível de dispositivos, mensagens entre processos TCP/IP ou UDP, e gerenciamento de pacotes. Dentre todas, a mais importante característica de ROS é a presença de uma grande comunidade de pesquisadores que contribui para a expansão do software, que é possível, principalmente, pela repositório disponível no website ROS.

ROS, como o próprio autor [41] salienta, não é um sistema operacional (SO) no sentido tradicional de gerenciador de processos e agendamento, mas apresenta algumas ferramentas semelhantes, como mensagens entre processos. De acordo com o autor, a filosofia por trás do desenvolvimento do ROS pode ser resumida nos seguintes pontos:

- Arquitetura de rede ponto-a-ponto (*peer-to-peer*).
- Tools.
- Multi linguagens de programação.

- *Thin.*
- Gratuito e aberto.

Um sistema desenvolvido em ROS é capaz de possuir diversos processos, em diferentes hosts, conectados por uma topologia ponto-a-ponto, em tempo de execução. Uma topologia com servidor central começa a se tornar problemática se computadores são conectados por uma rede heterogênea. ROS necessita de um mecanismo de busca para permitir que processos se encontrem em tempo de execução, chamado *service* ou *master*.

O sistema se comporta de maneira semelhante ao Player em relação à conexão por soquete TCP/IP, possibilitando o uso de múltiplas linguagens, como Python, C++, Octave e LISP.

Em ROS, pequenas *tools* são usadas para construir e executar os componentes, em vez de um grande monolítico ambiente de desenvolvimento. As *tools* realizam diversas tarefas, como navegar pela árvore de código, obter e definir parâmetros, visualizar a conexão ponto-a-ponto, medir a largura de banda utilizada, representar graficamente dados, gerar automaticamente documentações, e outros.

A ideologia *thin* de ROS é possibilitar que códigos, como drivers e algoritmos, possam ser reutilizados em outros *middlewares*, fora de ROS. Alguns exemplos são algoritmos de visão de OpenCV, e algoritmos de planejamento de trajetórias do OpenRAVE.

Os elementos fundamentais de ROS são nós, mensagens, tópicos e serviços.

Nós são processos ou módulos de software no código de controle, por exemplo um nó *camera* poderia processar todos os dados visuais. Os nós podem se comunicar com outros nós por simples mensagens (de tipos *integer*, *floating point*, *boolean*, *strings* etc), que são enviadas a um ou múltiplos tópicos (tipo *strings*). Um nó interessado em certo tipo de mensagem deverá subscrever ao tópico apropriado. Pode haver múltiplos publicadores e subscriptores concorrentes a um mesmo tópico.

Apesar de o modelo de comunicação do tipo editor-subscritor ser flexível, sua rotina de *broadcast* não é apropriada para transações síncronas. Para trocas de mensagens síncronas em ROS, utiliza-se serviço (também tipo *string*) um par de mensagens: um para pedido, outro para resposta, análogo à URI de um web service.

A figura 2.26 mostra os tipos de comunicação do sistema ROS.

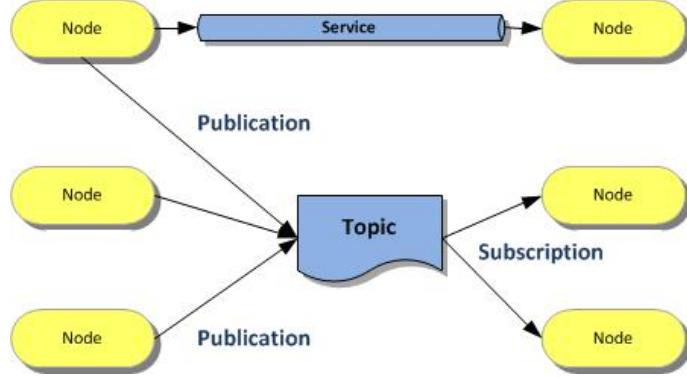


Figura 2.26: Tipos de comunicação no sistema ROS.

A figura 2.27 é um resumo de todas as características discutidas acima, mostrando como funciona o ambiente de desenvolvimento ROS. Há um computador na base, rodando nós de ROS, e um computador embarcado no robô, também rodando nós de ROS. Os componentes no robô são drivers que se comunicam com os diversos hardwares do robô (sensores e atuadores), algoritmos para processamento (de imagens e outros), e controladores dos atuadores do robô (controle PID e outros). Na base, componentes mostram os status do robô para o usuário e exercem a função de visualização. Está sendo representada tanto a comunicação tipo *service* (servidor-cliente), quanto a comunicação editor-subscritor (*broadcast*).

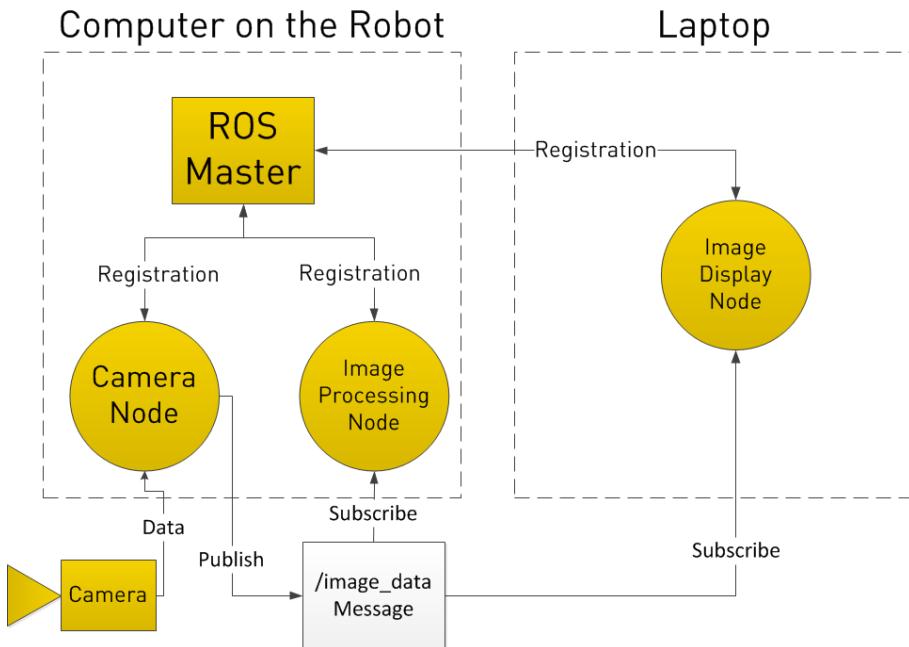


Figura 2.27: Arquitetura do ambiente de desenvolvimento ROS.

As vantagens da utilização de ROS no desenvolvimento de sistemas robóticos

são diversas, como comentado em [42]. ROS é amplamente utilizado como *framework* para a camada funcional de uma arquitetura híbrida. É um sistema flexível e que possui um repositório crescente de pacotes de software no estado da arte. Esses pacotes fazem interface com diversos hardwares robóticos, e algoritmos que realizam diferentes tarefas do robô, como SLAM, processamento de imagem e etc. O acesso a este repositório facilita e agiliza a prototipagem e o desenvolvimento de complexos sistemas robóticos. Os módulos da camada funcional, nós de ROS, podem ser programados em diferentes linguagens, como Python, C++ e Java. Os módulos podem ser iniciados, interrompidos e reiniciados em tempo de execução, e se comunicam entre si numa rede *peer-to-peer*. Há duas formas de comunicação entre módulos: o estilo síncrono servidor-cliente e o assíncrono *publisher-subscriber*. Além disso, ROS suporta simuladores robóticos como Stage, Gazebo, MORSE e OpenRAVE.

2.5.2 Camada Executivo

Como já definido na subseção 2.4.2, a camada Executivo usa técnicas de planejamento reativo para selecionar os comportamentos (nós, ou componentes, em uma linguagem ROS) que entrarão em execução de uma biblioteca de componentes existentes. Ela sequencia a execução dos componentes para a realização de um certo objetivo, monitora o desempenho, traduz os planos de alto nível da camada Planejador em tarefas, executando-os apropriadamente. Na arquitetura implementada, a camada Executivo também aloca e monitora os recursos (gerenciamento de recursos).

Como já comentado na subseção 2.3.2, apesar de possuírem inúmeras vantagens, a arquitetura reativa possui problemas de *situatedness* e escala, os quais a camada Executivo deve se preocupar em resolver.

O problema de escala ocorre devido ao número de interconexões de uma arquitetura reativa, o qual aumenta exponencialmente em relação ao aumento de comportamentos. Isto ocorre, pois um comportamento obtém os dados de outros através de interconexões (um comportamento de nível 1 necessita estar interconectado a outros de nível 0 para obter os seus dados). Este problema já havia sido resolvido por uma arquitetura com comunicação do tipo *blackboard* [38], onde os dados dos sensores são disponibilizados a todo os componentes, pois há um componente central que

armazena os dados e envia a todos que necessitam. Em ROS, a comunicação de estilo *publish-subscriber* resolve o problema, sem a necessidade de implementação extra, já que todos os componentes podem “ouvir” os dados publicados por outro componentes.

Alguns comportamentos devem ser executados apenas em situações específicas, não durante todo o funcionamento do robô. O problema de *situatedness* é a execução de todos os comportamentos ao mesmo tempo, mesmo quando estes não são necessários. Apesar de estarem sendo executados, não são percebidos na saída do robô, pois estão sendo suprimidos por comportamentos de nível superior (subsunção). O comportamento *desvio de obstáculos*, por exemplo, só faz sentido quando o robô está em movimento, e o comportamento *toque com manipulador* só fará sentido quando o robô está próximo de um equipamento.

Outro problema da arquitetura de subsunção é a *ciência da falha (cognizant failure)*, isto é, comportamentos não sabem quando seus processos estão em falha e continuam a execução sem resultados promissores. A camada Executivo deve reportar a falha à camada Planejador (*execution monitoring*), e suspender sua execução (*error recovery*).

O Sequenciador da camada Executivo é, normalmente, um controlador hierárquico de máquinas de estados ou Redes de Petri [43]. Algumas linguagens foram desenvolvidas para ajudar programadores na implementação desta camada: *Reactive Action Packages* (RPAs) [44], *Procedural Reasoning System* (PRS) [45], *Execution Support Language* (ESL) [46], *Task Description Language* (TDL) [47], *Plan Execution Interchange Language* (PLEXIL) [48]. Como a camada Funcional foi desenvolvida em ROS, o autor optou por analisar camadas Executivo integradas ao sistema ROS: Petri Net Plans (PNP) [49], e SMACH [50].

Todas as linguagens mencionadas provêem suporte à decomposição hierárquica de tarefas e subtarefas, capacidades de expressar condicionais e iterações, e suportam restrições temporais seriais ou paralelas entre tarefas (por exemplo, tarefa B deve começar após 10s do término da tarefa A - serial). Com exceção de PLEXIL, todas suportam recursividade de tarefas. Com exceção de TDL, as linguagens provêem suporte explícito para codificação de pré e pós-condições de tarefas e para especificação de critério de sucesso. ESL, PLEXIL e SMACH suportam sinalização

de eventos (avisam quando há transição de tarefas). ESL, TDL, PNP e SMACH suportam terminação de tarefas baseado na ocorrência de eventos (por exemplo, quando a tarefa B termina, A deve começar). As linguagens RAPs, PRS e ESL incluem uma base de dados simbólica (modelo do mundo) que se conecta aos sensores diretamente ou à camada Funcional, a fim de manter um modelo síncrono do mundo.

As linguagens lidam de maneira diferente em relação a monitoramento de execução e recuperação de erros. ESL, TDL, SMACH e PNPs provêem monitoramento explícito de tarefas, verificam erros com registros (como feito nas linguagens C++ e Java), e suportam procedimentos de “limpeza”, quando as tarefas são terminadas. RAPs e PLEXIL retornam valores para indicar falha, e não possuem verificação de erros. PRS suporta monitoramento da execução, mas não verificação de erros. ESL e PRS provêem a noção de recursos compartilhados e gerenciamento de recursos.

Detalha-se, em seguida, as camadas Executivo integradas ao sistema ROS.

Petri Net Plans (PNP)

Petri Net Plans [51] é uma linguagem baseada em Redes de Petri (*Petri Nets - PN*), desenvolvida em 2006 com integração ao sistema ROS. A linguagem permite o projeto de comportamentos intuitivos e eficientes para multi-robôs. Suas principais características são a possibilidade de concorrência e interrupção de tarefas, análise de planos baseado em ferramentas padrão de PN, e o desenvolvimento de comportamentos de maneira distribuída para multi-robôs garantindo coordenação e colaboração. O autor do sistema comenta que PNP é a primeira abordagem sistemática e metodológica para projetar planos baseados em PNs.

A grande vantagem da metodologia de projeto da camada Executivo em PNP é a exploração das técnicas e ferramentas de análise de planos das PN. PNP é introduzido como um subconjunto de PNs, baseando-se em primitivas de modelagem de linguagens de ação inspirada por Situation Calculus [52], como ConGolog [53]. A linguagem resultante é mais expressiva que a maioria das abordagens na literatura, permitindo formas complexas de sensoriamento, laços, interrupções, concorrência, coordenação e cooperação. Comparando-se com sistemas baseados em máquinas de

estados (FSM), as PNs são geralmente exponencialmente mais compactas, já que, por exemplo, uma PN finita pode representar uma FSM infinita.

Dentre as diversas aplicações realizadas com PNP, destacam-se: um robô com rodas usado para busca e missões de resgate; e quatro robôs AIBO utilizados para o futebol de robôs.

Apesar das inúmeras vantagens, a linguagem ainda não foi utilizada em robôs que exijam robustez. Além disso, a interface com ROS não é perfeitamente estável, e ainda não é possível executar PNs em paralelo. A construção das PNs ainda requer software externo independente (JARP).

SMACH

Em 2011, Bohren testa SMACH, camada Executivo, no robô PR2, integrado ao sistema ROS [54]. PR2 é um robô planejado a executar uma tarefa com grande repetição e robustez, em um ambiente bem definido, numa aplicação que requer interação com o ser humano, e que há métricas de sucesso ou falha: servir bebidas (isto é, localizar, manusear e entregar). O sistema é completamente autônomo e integrou capacidades como: identificação dinâmica de obstáculos, identificação de geladeiras, tipos de bebidas, e faces humanas. Componentes de planejamento incluem: navegação, planejamento de movimentos (*Motion Planning*) para manipulador com objetivo e restrições de trajetória, e módulos de *grasping* (segurar objetos).

Há três tipos de componentes na arquitetura do PR2:

- *subsistemas de propósito geral*: contêm códigos que não são específicos à aplicação e podem ser usados por outras. Exemplos: controle de *grasping* de manipulador, controle de trajetória de manipulador, cinemática inversa de manipulador, controle de força e velocidade de manipulador, planejamento de trajetória não-holonômico, planejamento de trajetória holonômico, sensor laser, sensor das juntas do manipulador, sensor de áudio, e etc.
- *código de aplicação específica*: representa a coleção de processamento de dados de sensores e ações de alto nível que são particulares à aplicação. Exemplo: camada Executivo, abrir porta, pegar bebida, fechar porta, entregar bebidas, detecção de rostos, e outros.

- *interface de usuário.* Exemplo: visualizador SMACH, RViz (visualizador integrado ao ROS), interface Web, e outros.

O PR2 utiliza um novo *framework* de camada Executivo, o qual permite a implementação de sequência de tarefas “bem definidas”, isto é, a operação nominal, modos de falha e sequências de recuperação de falhas podem ser descritas explicitamente. O novo *framework* é uma biblioteca independente de ROS, desenvolvida em Python, chamada SMACH. SMACH provê estruturas para geração procedural de programas baseados em máquinas de estado concorrentes em hierarquia. Além disso, apesar de ser uma biblioteca independente de ROS, provê módulos de integração como mensagens, tópicos e ações *actionlibs* de ROS.

SMACH pode ser usado para a construção e execução hierárquica de máquinas de estado concorrentes (o formalismo pode ser encontrado em *statecharts* [55]).

A) Características do SMACH

As características do sistema SMACH são aqui detalhadas, já que a implementação da arquitetura da DORIS também utiliza SMACH como parte de sua camada Executivo.

- 1) *Structures* (Estruturas): há duas interfaces primárias definidas por SMACH:

- ***State*** (Estado): uma interface para um objeto representando um *state of execution* (estado de execução) com um conjunto potencial de *outcomes* (resultados). *States* só necessitam definir uma única função *execute*, que o bloqueia até retornar um *outcome* para aquele *state*. *Outcomes* de *states* podem ser pensados como uma “interface” para um dado *state* e devem ser definidos durante a construção. Um *outcome* exemplo de um estado seria: *succeeded* (sucesso), *aborted* (abortado) ou *preempted* (colocado em espera devido a nova tarefa).
- ***Container***: uma interface para um objeto representando uma coleção de um ou mais *states*. *Containers* definem diferentes políticas de execução baseadas em seus *states* filhos e *outcomes*.

O SMACH *container* mais simples é o *StateMachine* (Máquina de Estado). Uma máquina de estado SMACH pode ser entendida como diagramas de fluxo de estados, onde nós são estados em execução (o robô fazendo algo) e arestas representam

transições de um estado para outro via um *outcome*. Estes *containers* podem ser compostos hierarquicamente e podem ter seus próprios *outcomes*.

SMACH também provê um *container* simples do tipo *split-join* (dividir-juntar), o *Concurrence* (Concorrente). Diferente do *StateMachine*, que executa um *state* por vez em série, o *Concurrence* executa mais de um *state* simultaneamente. Os *outcomes* de um *Concurrence* podem ser definidos em sua construção. O *Concurrence* padrão do SMACH só termina quando todos os seus filhos terminam, mas isso pode ser configurado.

2) *User Data* (dados do usuário): os planos construídos em SMACH possuem certas capacidades não disponíveis em máquinas de estados finitos formais. Cada *container* SMACH tem um dicionário de escopo local de dados de usuário que pode ser acessado por seus estados filho. Estados podem armazenar e carregar dados desta estrutura em tempo de execução. Como os *outcomes*, estas entradas e saídas podem ser consideradas parte da interface para um estado e devem ser declarados durante a construção. *Containers* podem, também, possuir entradas e saídas chave (*input and output keys*), a fim de mover dados entre escopos.

Apesar de esta característica deixar a análise dos estados mais complexa, ela deixa o sistema muito poderoso. Ela permite que dados sejam passados entre estados consecutivos, e, além disso, os dados podem ser acumulados por vários estados de forma que estejam disponíveis, no futuro, a um ramo durante a execução. Isto significa que um estado “completo” de uma árvore SMACH, em um determinado momento, é a união dos estados ativos do nível de execução em cada *container* e os conteúdos de dicionários do *user data* em cada *container*. A figura 2.28 mostra um *container* exemplo do SMACH, chamado *SM_TOP*, que possui dois estados *FOO* e *BAR*, tal que as entradas do *container* *SM_TOP* (*input keys*) são recebidas pelo estado *FOO*, os dados de saída de *FOO* são entradas do estado *BAR* e os dados de saída de *BAR* são os dados de saída do *container* *SM_TOP*.

B) Prototipagem e visualizador

SMACH foi projetado para permitir prototipagem rápida e intuitiva de aplicações robóticas. A biblioteca de interface SMACH-ROS provê a visualização, em tempo de execução, dos planos criados em SMACH, o SMACH Viewer. O visualizador SMACH renderiza a estrutura dos planos, realça os estados sendo executados em

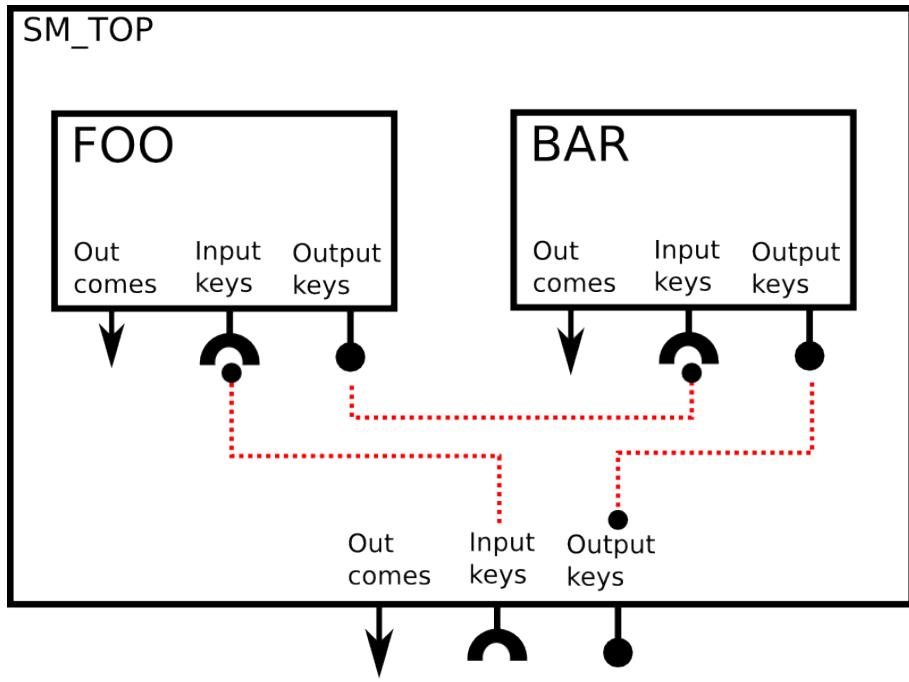


Figura 2.28: Container *SM_TOP* que possui dois estados e transições de dados.

tempo real, e lista os conteúdos do dicionário de dados para um dado *container*.

Esta ferramenta permite ao desenvolvedor:

- Descobrir pontos de falha e rapidamente adicionar novos estados que possam estimar a natureza da falha ou para recuperação.
- O SMACH Viewer é suficientemente intuitivo, de forma que o programador rapidamente encontra erros.
- A estrutura é bastante modular, logo os processos que são executados no robô podem ser divididos para múltiplos programadores sem conflito.
- SMACH pode ser executado a partir de qualquer ponto do plano sem alteração no código.
- SMACH possui propagação de *preemption*, isto significa que um pedido de cancelamento vindo do nível superior da árvore de execução atua no estado ativo e propaga a ação de cancelamento por todos os estados da folha, cancelando tarefas concorrentes de maneira segura.

C) Padrões de projeto (*Design Patterns*)

Padrão de projeto é uma solução geral reutilizável para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software. Uma camada Executivo pode usar o **Sequenciador** hierárquico de máquinas de estados do SMACH para estruturar um plano, mas este *framework* não é uma camada Executivo completa, pois há outras responsabilidades desta camada (seção 2.4.2). SMACH utiliza padrões de projeto de ROS para suprir a responsabilidade de monitoramento de execução.

1) Coordenação de dados: os estados em SMACH podem ser ações do tipo *actions*, implementadas no ambiente ROS. Esse tipo de ação recebe uma mensagem objetivo e retorna mensagens de resultados e status (*succeeded*, *aborted* e *preempted*).

2) *Preemption* mútua e monitoramento de tópicos: *containers* concorrentes podem ter diversas políticas de terminação definidas como funções de terminação. Um padrão de projeto é o *preemption* mútuo de duas tarefas concorrentes, útil quando o término de uma ação deveria causar o término de outra.

D) Aplicações

Além da demonstração de desempenho na utilização de SMACH em PR2 [54], há outros sistemas que já fazem uso desta camada Executivo. A dissertação de mestrado [56] implementa uma camada Planejador em cima de SMACH. O *Deutsches Forschungszentrum für Künstliche Intelligenz* (DFKI), um dos maiores institutos sem fins lucrativos de pesquisa no campo de inovação de tecnologia de software e robótica baseado em métodos de inteligência artificial, desenvolveu um robô móvel autônomo de dois braços, AILA [57], que usa SMACH como camada de nível Executivo. E já há outros sistemas que se baseiam na filosofia SMACH, como em [58].

2.5.3 Camada Planejador

Na subseção 2.4.2, o Planejador é definido como uma camada composta pelo planejador de missão e o cartógrafo. A camada determina as atividades do robô de longo prazo baseada em objetivos de alto nível. Enquanto a camada Funcional possui responsabilidades de com o aqui e agora, e a camada Executivo com o que acabou de acontecer e o que deve acontecer depois, a camada Planejador olha para o futuro. Nesta subseção, aborda-se as RDEs voltadas para a camada Planejador, como elas funcionam internamente, e destaca-se o Rapyuta, uma camada Planejador visionária

que está sendo integrada ao ambiente ROS.

De acordo com [32], as duas abordagens mais comuns são o planejador de rede de tarefas hierárquicas (*hierarchical task net* - HTN) e o planejador/agendador. Os HTN, como o O-Plan (*The open planning architecture*[59]) e o SHOP (*simple hierarchical ordered planner*)[60], decompõem tarefas em subtarefas de uma maneira similar às camadas Executivo. A diferença é que os HTN operam em um alto nível de abstração, usam a informação dos recursos disponíveis, e possuem métodos de lidar com conflitos entre tarefas (tarefas que necessitam do mesmo recurso, ou uma tarefa negando uma pré-condição necessária por outra tarefa). A base de conhecimento do robô é o arquivo (dicionário) de tarefas.

O planejador/agendador, como em [61] e [62], são úteis em aplicações onde tempo e recursos são limitados. A camada Planejador cria planos de alto nível que agenda quando tarefas devem ocorrer, mas tipicamente deixa ao Executivo a responsabilidade de determinar exatamente como alcançar os objetivos. Planejador/agendador organizam as tarefas e os recursos do robô cronologicamente e com prazos a fim de evitar conflitos (motores, energia, comunicação e outros). A base de conhecimento do robô inclui os objetivos das tarefas, os recursos necessários, suas durações, e restrições entre tarefas.

Rapyuta

Rapyuta é a *RoboEarth Cloud Engine*, uma Plataforma como serviço (*Platform-as-a-Service* - PaaS) de código aberto, serviço de hospedagem e implementação de hardware e software, usado para prover aplicações por meio da internet e projetado especificamente para aplicações robóticas. A grande motivação é o número cada vez maior de robôs saindo do ambiente industrial para o ambiente doméstico e praticando tarefas caseiras (como servir bebidas, dobrar toalhas e outras). Algumas tarefas domésticas, no entanto, exigem grande poder computacional, e os computadores embarcados cada vez consomem mais bateria, restringindo a mobilidade do robô, reduzindo tempo de operação e aumentando os custos. A computação interna do robô pode ser reduzida se algumas computações que não exijam tempo real puderem ser realizadas na nuvem, como planejamento de segurar objetos (*grasp planning*), mapeamento, e navegação.

Rapyuta é um projeto que busca resolver os desafios de construir uma plataforma robótica em núvem. O *framework* é baseado em um modelo de computação elástico [63] que aloca dinamicamente ambientes de computação segura para robôs. Esses ambientes de computação são interconectados, permitindo que robôs compartilhem dispositivos e informações entre si, tornando Rapyuta uma plataforma para multi-robôs. Além disso, os ambientes de computação provêem acesso com banda larga ao RoboEarth, repositório de conhecimento, permitindo que robôs tenham conhecimento da experiência de outros robôs. Os robôs podem submeter e pegar dados do repositório RoboEarth, e ainda processar planejamentos dentro da ferramenta.

Rapyuta é compatível com ROS, o que permite a execução de todos os pacotes de código aberto disponíveis em ROS.

Capítulo 3

Arquitetura proposta

O capítulo 2 apresentou os conceitos necessários para o entendimento desta dissertação, mostrou a evolução das arquiteturas robóticas e controles de missão, e as diversas aplicações em robôs modernos. A autonomia de um robô depende, em grande parte, do desenvolvimento desta arquitetura robótica.

Neste capítulo, será apresentada uma implementação da arquitetura híbrida de três camadas, utilizando o ambiente de desenvolvimento ROS, para um robô móvel que executa tarefas de inspeção, a DORIS. Como definido no capítulo 2, a arquitetura robótica não é apenas uma arquitetura de software, mas sim uma arquitetura que depende dos componentes físicos do robô (hardware) e a sua aplicação. Dessa forma, faz-se necessário apresentar o objeto de estudo desta dissertação.

3.1 Robô DORIS

DORIS é um robô desenvolvido pela COPPE/UFRJ, em colaboração com Petrobras e Statoil, para aplicação *offshore*: monitoramento e inspeção em plataformas de petróleo. O robô é controlado de maneira autônoma ou teleoperado, permitindo que o operador acompanhe o estado da missão, e monitore a informação dos sensores. O robô apresenta as seguintes funcionalidades [64]:

- Detecção de anomalias por vídeo: uso de múltiplas câmeras (luz visível, infravermelho, fisheye e estéreo) para detectar anomalias, como objetos abandonados, fumaça, fogo, intrusos e vazamentos.

- Detecção de anomalias por áudio: microfones detectam áudios anômalos, como explosões, e realizam diagnóstico de mau funcionamento de máquinas, comparando os sons recebidos com assinaturas obtidas previamente.
- Detecção de anomalias por vibração: DORIS possui um manipulador com sensor de vibração em seu efetuador para inspecionar o funcionamento de máquinas, a partir de algoritmos com classificadores de falhas.
- Detecção de anomalias por sensor de gás: sensor de hidrocarboneto detecta o vazamento de gases.
- Mapeamento 3D do ambiente: DORIS é capaz de reconstruir um ambiente 3D a partir de câmeras e Laser.
- Detecção de anomalias por temperatura: DORIS possui um sensor de temperatura e umidade.
- Detecção de anomalias por câmera de infravermelho: uma câmera de infravermelho pode detectar pessoas, indicar a presença de intrusos ou indentificar incêndios.

DORIS é um robô móvel que se locomove em um trilho pelo uso de dois gimbals, os quais contêm atuadores e rodas. O robô foi desenvolvido dentro da filosofia de modularidade, isto é, novos sensores podem ser integrados ao sistema, ou até um novo módulo do robô pode ser adicionado, a fim de melhorar seu desempenho dentro do escopo da aplicação de inspeção. Isso exige a flexibilidade e modularidade do sistema mecânico, software e sistema elétrico/eletrônico.

O sistema é alimentado por quatro baterias, composto por um computador embarcado com processador de alto desempenho e memória, e um *solid-state drive* para armazenamento. Possui comunicações: Wireless IEEE 802.11n com a base (operador); *Controller Area Network* (CAN) entre computador embarcado e drivers dos atuadores; rede *Local Gigabit Ethernet* para os diversos componentes internos do robô; e rádio 2.4/5.0 GHz para emergência. Possui atuadores: quatro motores 200 W EC-4pole para locomoção; e quatro motores para as juntas do manipulador. Sensores: câmera fixa; câmera térmica; câmera *fisheye*; duas webcams; e uma *Inertial Measurement Unit* (IMU). Além disso, há um sistema eletrônico de suporte

de veículo, chamado *Vehicle Support System (VSS)* [65], capaz de detectar falhas eletrônicas, distribuir energia de maneira ótima entre os componentes e proteger o robô em situações emergenciais.

O VSS possui funções que independem do software e da arquitetura robótica, são considerados de alto risco e, portanto, não estão disponíveis para programador e usuário. Em detalhes, as funções do VSS são:

- Detecção de falhas: em dispositivos, pelo monitoramento de corrente/tensão; no módulo, pela verificação de temperatura/umidade;
- Proteção de dispositivos contra sobrecorrente graças a fusíveis;
- Distribuição ótima de energia e monitoramento de baterias;
- Botão de emergência para desligamento manual ou via rádio.

O trilho por onde o robô se desloca é construído a partir de tubos de policloreto de polivinila (PVC) e possui seções retas, curvas ortogonais de subida, descida, para a direita e para a esquerda. Os segmentos curvos são tubos de 1 m dobrados 90°, resultando em curvaturas de aproximadamente 630 mm. Um trilho para testes foi construído no Centro de Pesquisas e Desenvolvimento da Petrobras (CENPES) e sua extensão é cerca de 140 m. As figuras 3.1 e 3.2 ilustram o robô, o trilho e o ambiente em que trabalha.



Figura 3.1: A DORIS e o trilho.



Figura 3.2: A DORIS no CENPES.

3.2 A arquitetura robótica implementada

Na seção 3.1, foi apresentada a DORIS e suas funcionalidades. Como foi definido na seção 2.1, pela visão do autor, o conceito de *arquitetura robótica* não é equivalente a uma arquitetura de software, é necessária a avaliação do robô como um todo, isto é, seus elementos de hardware e sua aplicação. O desenvolvedor de ROS [41] já afirmava que não há melhor RDE, já que todos apresentam vantagens e desvantagens, e o mesmo pode ser afirmado no contexto mais amplo de arquitetura robótica e em paradigmas. A arquitetura ideal para um robô pode ser ineficiente para outro. A arquitetura robótica desenvolvida tenta ser geral para robôs com características e desafios semelhantes ao DORIS, como AUVs e UGVs.

DORIS exigem grande processamento de imagem (câmeras) e de outros sensores. O usuário final poderá, com o auxílio de uma interface, programar as missões: inspeção até uma posição do trilho, inspeção de equipamento, ir até uma posição do trilho, e ronda (inspeção global). Além disso, o usuário pode tomar o controle manual do robô, e visualizar as respostas dos sensores.

As funcionalidades de inspeção do robô exigem a construção e manutenção do modelo do mundo, pois há constante comparação com a situação esperada e possíveis anomalias. Também fica claro que não é só um modelo do mundo que se faz necessário, mas a construção de modelos distintos para cada componente de forma a

otimizar as missões. Além disso, robôs *offshore* exigem grande robustez, por trabalharem em ambientes hostis, e que demandam ações rápidas em situações emergenciais. As exigências de um controle de missão para modos flexíveis de inspeção, a manutenção de modelos de mundo e a robustez no controle do robô demandam uma arquitetura híbrida.

No capítulo 2, foi explicitada algumas formas de arquitetura híbrida, mas aquela com maior número de aplicações bem sucedidas é a arquitetura híbrida de três camadas, utilizadas desde a aplicação do carro autônomo do desafio DARPA (Stanley) até a aplicação de exploração do robô da NASA no planeta Marte.

Escolhida a arquitetura híbrida de três camadas como arquitetura robótica da DORIS, ainda é necessário desenvolver a implementação de cada camada, isto é, como e qual executará as funções descritas por Murphy: sequenciador, gerenciador de recurso, cartógrafo, controle de missão, e monitor de desempenho. A arquitetura de três camadas é formada por: camada Funcional, camada Executivo e camada Planejador, e como cada arquitetura de três camadas apresenta suas peculiaridades, as camadas da DORIS são detalhadas adiante.

3.2.1 Implementação da camada Planejador

As diversas arquiteturas robóticas de três camadas diferem em relação às responsabilidades da camada Planejador. Kortenkamp [32], por exemplo, aponta que as responsabilidades mais comuns da camada Planejador são criar redes de tarefas hierárquicas, ou agendamentos e gerenciamento de recursos. Em [5], o Planejador pode assumir responsabilidades de Adaptador, ou Seletor, ou Conselheiro (ver seção 2.4.2). A camada Planejador implementada na DORIS terá as seguintes responsabilidades:

- Controle de missão: como definido na subseção 2.1, é a interface com o usuário, permitindo que o operador defina as missões. Além disso, o controle de missão traduz os comandos de missão do usuário ao robô e provê feedback ao operador (*execution monitoring*).
- Agendador: é a responsabilidade de agendar as missões, e reorganizá-las quando não há um recurso disponível ou em caso de falha.

- Cartógrafo: é a responsabilidade de criar, e realizar a manutenção dos mapas, ou modelos do mundo. Além disso, deve gerar novos mapas para otimizar as diversas funções executadas no robô, quando houver necessidade.

Muitas arquiteturas especializadas em planejamento de trajetórias colocam esta responsabilidade também para a camada Planejador, outras a separam em duas camadas, como planejamento global e planejamento local. Na DORIS, não há um planejamento de trajetória, já que o trilho é um mapa unidimensional, porém há um planejamento de velocidades (parte do planejamento de movimento - *Motion Planning*) atribuído à camada Funcional.

Antes de detalhar a camada Planejador, é necessário definir alguns termos específicos para melhor entendimento desta dissertação.

Definição: **Plano de Missão** ou **plano** é o conjunto de **missões** definidas ou escolhidas pelo usuário, em uma interface, a serem realizadas sequencialmente pelo robô.

Definição: **Missão** é o comando básico, com alguns parâmetros a serem definidos (*argumentos*), que o usuário pode dar ao robô, em uma interface. A **missão** é um conjunto de tarefas.

Definição: **Tarefa** é um processo executado pelo robô com um determinado objetivo. O conjunto de tarefas ordenado sequencialmente compõe a missão. Uma tarefa pode ser cancelada por outras tarefas (subsunção).

Definição: **Estado** é o processo básico definido na camada Executivo SMACH. Em linguagem de arquitetura robótica, esta definição se confunde com **comportamentos** por ser um processo básico e ser modelado como uma AFSM, porém é uma definição mais geral por não necessariamente envolver um par estímulo-ação/sensor-atuador. Um conjunto de estados com um fim compõe uma tarefa.

A figura 3.3 mostra a arquitetura da camada Planejador, destacando suas responsabilidades e suas interconexões entre camadas.

Controle de missão

O controle de missão é a combinação de três responsabilidades interligadas: a interface com o usuário, o acompanhamento e feedback das missões ao usuário e a tradução das missões ao robô. Portanto, o controle de missão é a parte da arqui-

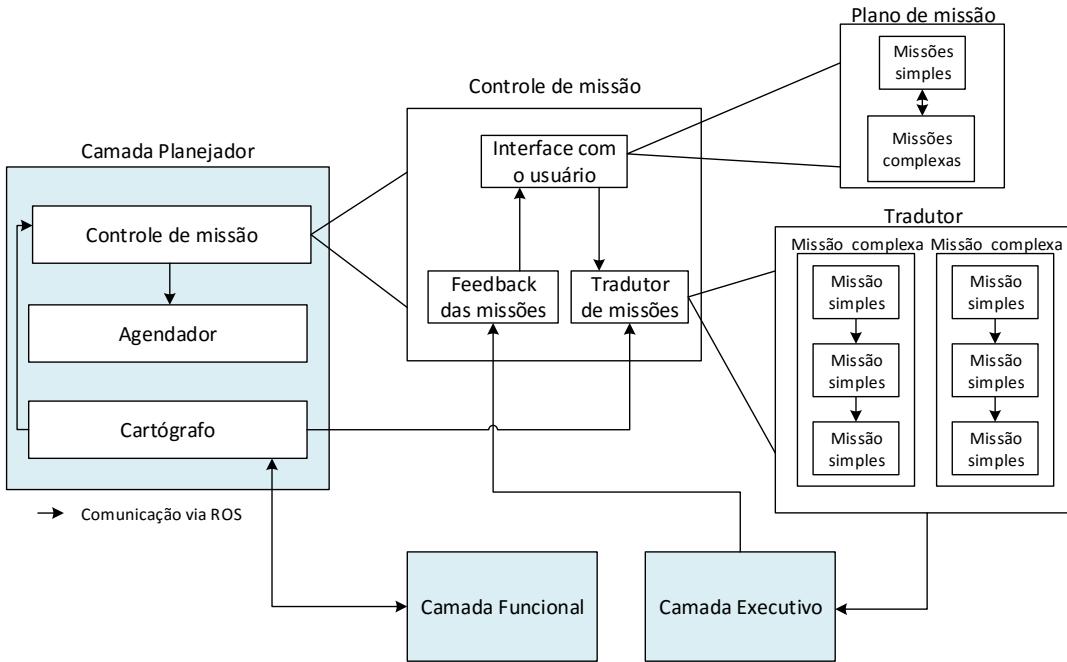


Figura 3.3: Arquitetura da camada Planejador.

tetura robótica que faz a interface Homem-Robô: traduz a linguagem do homem à linguagem do robô e vice-versa.

A interface de usuário da DORIS possui o conjunto de missões e seus parâmetros disponíveis para o operador. O usuário pode adicionar missões sequencialmente, salvar um plano de missão, agendá-lo e/ou executá-lo (botão *Play*). Além disso, a interface disponibiliza os logs das missões.

A camada Planejador enviará missões à camada Executivo, quando estas estiverem agendadas ou quando o usuário requisitar. Na subseção 2.4.2, foi exemplificada a **tradução** da camada Planejador com um robô assistente que serve cafés. A implementação desta responsabilidade na DORIS é realizada por um dicionário codificado no robô. Este método é normalmente conhecido na literatura como *encoded knowledge* (conhecimento codificado), que são dados incorporados ao código fonte (*hard coded*).

As missões do robô podem ser classificadas em: **Missões Simples**, **Missões Complexas** e **Missões Desconhecidas**. Todas as missões possuem *argumentos*, entradas (*inputs*) mínimas necessárias para a execução da missão.

As missões simples não necessitam de tradução por já estarem modeladas em tarefas e máquinas de estados SMACH. Por exemplo:

- **GOTO(x,vel,direction,lap,circular)**: missão de locomoção do robô até um ponto do trilho. Os *argumentos* para esta missão são: posição desejada no trilho, em valor absoluto; velocidade desejada, um valor fuzzy: “slow” (devagar), “normal” (normal) ou “fast” (rápida); direção de movimento, 1 para sentido de locomoção para frente (baterias chegam por último) e -1 para sentido contrário; número de voltas desejado, em caso de trilho circular, antes de o robô atingir a posição requisitada. O único argumento obrigatório é posição desejada, de forma que os valores padrão são: velocidade “fast”; direção de movimento de distância mínima; zero voltas; e trilho circular.
- **RECORD(type)**: gravar dados de sensores no robô. Há um *argumento* para esta missão que pode ser preenchido como: vídeo, áudio e termografia. Cada tipo representa uma missão de gravar dados de um sensor específico (câmera, microfone e câmera termográfica, respectivamente).
- **STOP_RECORD()**: parar gravação de dados de sensores no robô.
- **DETECTING_ANOMALIES(type)**: iniciar algum detector específico de anomalia. Há três tipos de detecção de anomalias que podem ser escolhidos: vídeo, áudio e termografia. Para cada um deles, há uma máquina de estados específica: iniciar algoritmo detector de anomalias por câmera, microfones ou câmera termográfica, respectivamente.
- **STOP_DETECTING_ANOMALIES()**: finalizar algum detector específico de anomalia.

As missões complexas, de mais alto nível, são combinações de missões simples. Estas necessitam ser traduzidas pelo Planejador com os argumentos corretos. Por exemplo:

- **GOTO_BASE()**: missão de locomoção do robô à base. Não há *argumentos* para esta missão. Tradução necessária para camada Executivo: **GOTO(0,’fast’)**.

- **INSPECTION(x,type)**: missão para inspecionar um trecho do ambiente. *Argumentos*: até onde o robô deve inspecionar (x - posição final absoluta no trilho); e o tipo de inspeção: vídeo, áudio ou termográfico. Tradução: **DETECTING_ANOMALIES(type)**; **GOTO(x,’slow’)**: velocidade “slow” já que inspeção requer velocidade mínima; **STOP_DETECTING_ANOMALIES()**.
- **PATROL()**: realizar uma ronda completa, isto é, inspecionar todo o ambiente. Tradução: **INSPECTION(lap=1,types)** (todos os sensores, posição final do trilho).
- **MANIPULATOR_INSPECTION(equipment)**: inspecionar um equipamento com manipulador. O *argumento* desta missão é o equipamento a ser inspecionado. O robô deve ir à posição do equipamento, tocar o equipamento com o sensor de vibração através do manipulador, e executar uma detecção de anomalias por vibração. Em linguagem da camada Executivo, seria **GOTO(x,’fast’)**: posição do equipamento (requer verificar mapa de equipamentos, busca no cartógrafo); **MANIPULATOR_POSITION_CONTROL([x,y,z])**: posição referente ao equipamento; **MANIPULATOR_FORCE_CONTROL()**; **STOP_DETECTING_ANOMALIES()**.

As missões desconhecidas são missões que não estão descritas no robô, isto é, missões que, até o momento, não pertencem ao dicionário do robô. Até esta fase da implementação, DORIS não permite que o usuário use missões desconhecidas. O motivo principal disso é que o robô não interage com o ser humano de maneira direta, mas apenas através de uma interface gráfica. Vale, porém, explicitar situações de aplicações robóticas em que missões desconhecidas são essenciais:

1. Robô que serve bebidas: uma missão desconhecida é o usuário pedir uma bebida que não existe em sua base de conhecimento (mapa de bebidas). Durante a tarefa de reconhecer a bebida na geladeira, o robô irá falhar. Porém, o Planejador poderia executar uma busca por fotos de bebida em algum site de buscas, antes de executar a tarefa, e atualizar a sua base de conhecimento

(mapa de bebidas), de forma que a tarefa tenha resultado positivo com novo dado.

2. J.A.R.V.I.S., a inteligência artificial das histórias em quadrinhos Iron Man (homem de ferro): comandos por voz normalmente são complexos, já que há diversas maneiras de pedir um plano a um robô. Por exemplo, os planos “Robô, preciso de uma tesoura”, “Robô, dê-me uma tesoura”, “Robô, traga-me uma tesoura” são planos equivalentes, ditos de maneira diferentes. Todas estas diferentes formas devem estar disponíveis em alguma base de conhecimento, dentro ou fora do robô, como na nuvem (*cloud*), a qual deveria ser compartilhada com todos os robôs que exercem a mesma função. RoboEarth [66] é uma ideia visionária que está buscando criar uma *internet dos robôs* (como a *internet das coisas*), já está sendo integrada ao ambiente ROS, e sua contribuição é criar um repositório na nuvem com missões compartilhadas para todos os robôs que a utilizam.

Na interface, o usuário pode escolher sequencialmente um conjunto de missões e executar, por exemplo, o plano:

```
GOTO(30)  
  
RECORD('VIDEO')  
  
INSPECTION('AUDIO',50)  
  
STOP_RECORD()  
  
RECORD('AUDIO')  
  
GOTO_BASE()  
  
STOP_RECORD()
```

O controle de missão da camada Planejador traduz sequencialmente o plano nas missões simples:

```
GOTO(30)  
  
RECORD('VIDEO')
```

DETECTING_ANOMALIES('AUDIO')

GOTO(50)

STOP_DETECTING_ANOMALIES()

STOP_RECORD()

GOTO(0)

STOP_RECORD()

Os status, motivo de interrupções, e possíveis erros ocorridos de cada missão são fornecidos pela camada Executivo. O controle de missão tem acesso fácil a essas informações graças ao estilo de comunicação publish-subscriber, característica do *middleware* ROS. Dessa forma, o controle de missão subscreve o tópico de *Warnings*, interpreta as mensagens, quando necessário, e disponibiliza esta informação no espaço de logs da interface.

O controle de missão está implementado, mas ainda não embarcado no robô. A interface gráfica de usuário ainda não foi finalizada, mas esta pode ser simulado por mensagens de ROS, durante os testes.

Agendador

O usuário pode armazenar o plano de missões no robô com data e hora. A responsabilidade Agendador é verificar a hora agendada para a execução da missão, executar a missão na hora correta, e, caso esta não seja bem sucedida e o motivo do insucesso for contornável em tempo estimável, reagendar a missão. Com o Agendador, o robô ganha uma rotina, o que é importante a DORIS, um robô de inspeção.

Cartógrafo

Há diversas formas de armazenar mapas do mundo dentro do robô, como mapas geométricos, nívens de pontos, simbólicos, analíticos e outros. Cada forma apresenta vantagens e desvantagens, de forma que muitas vezes é necessário armazenar vários mapas para otimizar seus usos nas diversas funções do robô. No caso da DORIS, por exemplo, durante o planejamento de velocidades do robô, o mapa geométrico 3D do ambiente não é a melhor opção, já que a DORIS se movimenta em um espaço

unidimensional (o trilho), e carregar e processar um espaço 3D é muito custoso. A camada Planejador deve gerar um mapa unidimensional, ou um mapa simbólico, para otimizar a função de planejamento de velocidades. Além disso, a camada Planejador deve manter os mapas gerados atualizados.

Por ser um robô de inspeção, DORIS possui diversos mapas do mundo:

1. Frames de câmera de vídeo (matriz);
2. Frames de câmera termográfica (matriz);
3. Sonoro (analítico);
4. Seções do trilho (matriz);
5. Postes (ou outras características marcantes) do ambiente (núvem de pontos);
6. Scans de laser (núvens de pontos);
7. Equipamentos (matriz);
8. Distâncias do trilho ao solo (analítico);
9. Arquivo CAD do trilho (CAD);

Os mapas utilizados para inspeção são 1, 2, 3 e 6, e foram gerados com os sensores câmera fixa, câmera termográfica, microfones e laser, respectivamente. O mapa 9 é obtido durante a fabricação do trilho e não é mais alterado. O mapa 4 foi gerado pelo CAD da fabricação do trilho, ou seja, a partir do mapa 9, e será necessário para a locomoção do robô. Os mapas 5 e 8 são gerados a partir do mapa 6 e são necessários para a localização do robô. O mapa 7 é inserido manualmente pelo programador, sendo uma matriz que relaciona o equipamento às suas coordenadas cartesianas.

A camada Planejador ainda não integra, nem faz a manutenção e atualização dos mapas, já que isso requer integração de todos os algoritmos de SLAM do robô, o que foge do escopo desta dissertação. Entretanto, quando os algoritmos SLAM forem implementados e testados, é importante que esta responsabilidade seja finalizada na camada Planejador do robô.

3.2.2 Implementação da camada Funcional

A camada funcional foi a primeira implementada e exaustivamente testada na DORIS. Como nas outras arquiteturas de três camadas previamente apresentadas, a camada Funcional é o nível mais baixo, conecta sensores a atuadores, implementa controladores PID, e diversos outros algoritmos, por exemplo para a localização do robô. Utiliza-se Ubuntu/Linux como sistema operacional e ROS como RDE para a implementação do nível funcional, já que este apresenta todas as vantagens descritas na subseção 2.5.1, e linguagem de programação C++, a fim de se garantir maior eficiência computacional.

A camada funcional seguiu a recomendação dos desenvolvedores do ROS, sendo estruturada de maneira semelhante à camada funcional do sistema CLARAty. Neste, a camada funcional é uma interface com todo o sistema de hardware e suas capacidades. É um software orientado a objeto, obtendo assim modularidade de hardware, e uma estruturação apropriada para usar as propriedades de herança, em software. A figura 3.4 mostra a organização da camada Funcional da arquitetura CLARAty.

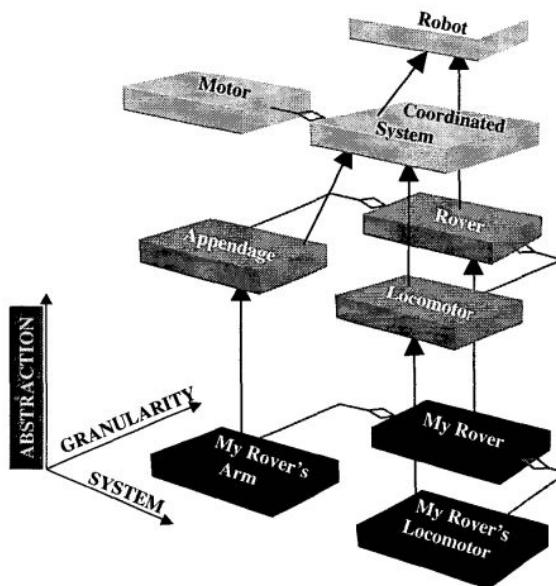


Figura 3.4: Organização da camada Funcional CLARAty

Na DORIS, foi proposto o *Robot Package Software*. Neste sistema, *Tools* (janelas gráficas) e *Componentes* (unidades de comunicação e processamento) são agrupados em *Robot Package* (bibliotecas dinâmicas). Os componentes que lidam com hardware rodam no computador embarcado do robô, já os componentes que inte-

ragem com as *tools* rodam no computador base. Os diversos componentes, tanto do computador embarcado quanto da base, se comunicam através de mensagens ROS, permitindo que a base controle o robô, na função de teleoperação [65]. Os componentes da base não fazem parte do escopo desta dissertação, já que não estão ligados à autonomia do robô, mas apenas a uma fração do controle de missão, que diz respeito à interface gráfica.

Dois *Robot Packages* fazem parte da DORIS: *General Package* e *DORIS Package*, derivado do primeiro. Outros robôs desenvolvidos por GSCAR, como o ROV LUMA, possuem seus pacotes específicos, derivados do *General Package*.

O *General Package* contém os componentes e *tools* gerais relacionados a vídeo, áudio, tabela de dados, *gamepad* (para controle remoto de robôs), e configurações de dispositivos que podem ser usados em outros robôs

O *DORIS Package* é um pacote mais específico à DORIS e lida com seus elementos de hardware e suas funcionalidades. Do ponto de vista de uma arquitetura robótica, neste pacote foram implementados os *comportamentos* específicos da DORIS. Os diversos componentes implementados para a DORIS foram divididos em funcionalidades, neste trabalho, a fim de proporcionar melhor entendimento.

Como já descrito na seção 3.1, os atuadores para locomoção e para o manipulador da DORIS se comunicam via CAN-Bus, que provê velocidade e transferência confiável dos dados [67]. Esta comunicação extra, exigiu a implementação do componente *CANOpen*, que utiliza a biblioteca SocketCAN disponível em Linux. Um componente *EPOS* é classe derivada de *CANOpen* e especifica este tipo de comunicação com o hardware EPOS (driver) utilizado na DORIS, além de implementar outras características do driver, como os dados dos encoders dos motores: posição (odometria), velocidade e corrente. Além disso, há a *EPOSNodelet*, classe derivada de *EPOS*, que implementa a comunicação ROS. A implementação dos componentes respeita a sugestão do desenvolvedor do *framework*, de forma que sempre há a criação de uma classe padrão C++ e uma classe derivada dentro do ambiente ROS, que usa os métodos do *framework*.

Os componentes que executam o controle de locomoção do robô são: *Controller*, *ControllerNodelet* (derivada de *Controller*), *MotionController* (derivada de *Controller*), *MotionControllerNodelet* (derivada de *MotionController*), *PositionController*

(derivada de *Controller*) e *PositionControllerNodelet* (derivada de *PositionController*). Os componentes *Controller* e *ControllerNodelet* são classes genéricas de controle, sendo necessária a implementação do controle para o robô específico, componentes *MotionController* e *PositionController*.

Os quatro hardwares EPOS possuem uma malha de controle PD ou PID para realizar o controle do robô por velocidade ou corrente, porém de maneira independente para cada motor. A sincronia e ajustes dessas malhas de controle são realizados no componente *MotionController*. O *MotionControllerNodelet* se comunica por serviço de ROS (*service*) com o componente *EPOSNodelet*, enviando os valores de velocidade desejados para cada motor de maneira síncrona. Por exemplo, observe que, em uma curva, devido à distância entre os dois gimbals do robô e à distância entre as rodas dos gimbals, os motores devem possuir velocidades diferentes. O *MotionControllerNodelet* envia *set-points* de velocidade às EPOS e pode alterar os parâmetros de aceleração e desaceleração. Além disso, este componente é responsável por receber entradas do componente *Joystick* e *Interface*, que podem controlar o robô pela base, ou *MissionController*, o componente de missão autônoma. Há apenas a prioridade do *MissionController* sobre todos os outros controladores, tal que, com exceção do *MissionController*, o componente que irá controlar será o primeiro que pedir o controle.

O componente *PositionController* será futuramente integrado ao *MotionController*. Sua funcionalidade é realizar um controle de posição, logo possui uma malha interna de controle, não disponível no hardware da EPOS, e se comunica com o componente *MotionController*, enviando *set-points* de velocidades. É um controle proporcional de posição com saturação de velocidade.

O manipulador da DORIS utiliza mais quatro EPOS (quatro motores) e exigirá ainda a implementação de um controle de força para sensoriamento de equipamentos ao toque, a partir de um sensor de vibração. Componentes como *ForceControl* e *InspectionVibration* ainda estão em desenvolvimento.

Ainda em fase de aperfeiçoamento, os componentes de localização *Localization* e *LocalizationNodelet* recebem mensagens de ROS (*subscriber*) do componente *EPOSNodelet*, os dados da odometria, isto é, quanto cada roda girou. Dentro de *Localization* é feita uma média para estimar quanto o robô se locomoveu. Em imple-

mentação, está sendo feito um sistema inteligente de localização com os dados dos sensores IMU e LaserScan. A fusão de sensores permitirá estimar de maneira precisa a posição do robô. O *LocalizationNodelet* ainda envia mensagens (*Publish*) para o *PositionController*, *MotionController* e *RVIZ*, um componente de visualização para o usuário.

Os diversos componentes de sensores, no robô, são: *AudioSender*, *AudioSenderNodelet* (derivada de *AudioSender*), *VideoSender*, *VideoSenderNodelet* (derivada de *VideoSender*), *VideoWebcamera* (derivada de *VideoSenderNodelet*), *AxisVideo* (derivada de *VideoSenderNodelet*), *LMS1xx*, *IMU*, *IMUNodelet* e *ColorDetector*.

O componente *AudioSender* é um driver que faz a interface com os diversos microfone disponíveis no robô. Comunica-se (*publisher*) por mensagem de ROS com o componente da base *AudioReceiverNodelet* (*subscriber*) para disponibilizar os dados ao usuário. Futuramente, irá se comunicar (*publisher*) com o *InspectionAudioNodelet* (*subscriber*), um componente que compara o áudio da base de dados do robô e detecta anomalias através de um algoritmo de reconhecimento de padrões.

O componente *VideoWebcamera* é um driver que faz a interface com as duas câmeras webcams disponíveis no robô. Comunica-se (*publisher*) com o componente da base *VideoReceiverNodelet* (*subscriber*) para disponibilizar os dados ao usuário, e com o componente *ColorDetection* (*subscriber*), um algoritmo que verifica a porcentagem de vermelho obtida em cada frame da câmera. Futuramente, o componente *ColorDetection* irá se comunicar (*publisher*) com o *Localization* (*subscriber*), já que a informação de vermelho no trilho será utilizada para calibrar a localização.

O componente *LMS1xx* é um driver que faz a interface com o LaserScan, sensor que realiza um escaneamento a laser do ambiente. Comunica-se (*publisher*) com o componente da base *LaserReceiverNodelet* (*subscriber*) para disponibilizar os dados ao usuário. Futuramente, irá também se comunicar com o *PoleDetection* (*subscriber*), que possui um algoritmo para detecção de postes, e o *Localization* (*subscriber*), já que este é mais um sensor que provê dados para o sistema de localização (altura do robô).

O componente *IMU* é um driver que faz a interface com a *IMU*. Ele envia uma lista de dados: velocidade, orientação, posição, pólos magnéticos e outros. Futuramente, também irá se comunicar com o *Localization*.

O componente *AxisVideo* é um driver que faz a interface com a câmera fixa da AXIS. Comunica-se (*publisher*) com o componente da base de mesmo nome para disponibilizar os dados ao usuário. Futuramente, o algoritmo de detecção de anomalias por vídeo será integrado ao sistema ROS, logo um componente *Inspection Video* será *subscriber* do mesmo tópico.

A figura 3.5 represente o esquema de comunicação entre os diversos componentes e suas hierarquias.

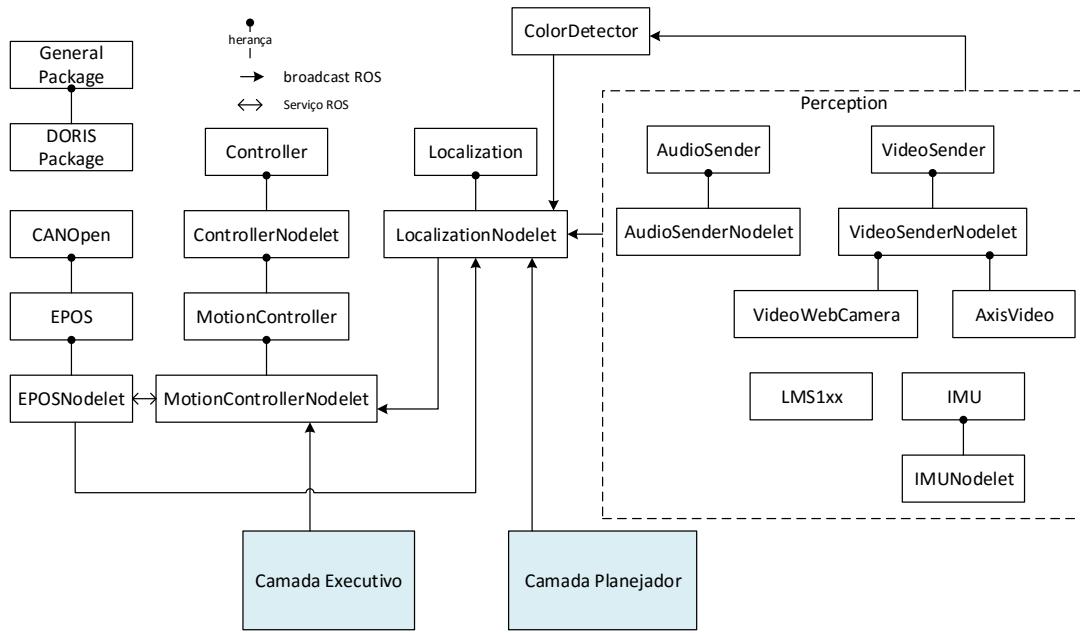


Figura 3.5: Camada Funcional da DORIS

Planejamento de velocidades

A DORIS se locomove em um trilho (mapa unidimensional), logo não há a necessidade de computação de um *Motion Planning* complexo. Quando uma missão do tipo **GOTO** é requisitada, apenas são calculadas as velocidades de segurança e compensação das rodas do robô, não a trajetória, pois só há uma trajetória possível, ou duas, caso o trilho seja circular. Apesar de o *Motion Planning* estar na camada Funcional, ele também é fundamental para a autonomia do robô, portanto o autor desenvolveu sua implementação.

O processo chamado *MasterPlanning* é um *thread* inicializado junto ao robô e

que subscreve ao tópico *DORIS/Vehicle/Path_Planning* para obter os parâmetros de entrada e publica sua saída ao tópico *DORIS/Vehicle/Path_Planning/Plan*. Suas entradas são: localização atual do robô; localização objetivo; direção de movimento (1 ou -1 - sentido natural ou sentido contrário); número de voltas desejado antes de chegar à localização objetivo; e se o trilho é circular (verdadeiro ou falso). Sua saída é uma matriz com n linhas e três colunas: a primeira coluna representa a posição do trilho onde ocorre troca de velocidade máxima; a segunda coluna é a velocidade máxima; a terceira é o *set-point* de posição.

A matriz de saída será utilizada pela tarefa *Motion_Planning_Position* da camada Executivo, a qual se comunica com o processo da camada Funcional *PositionController*. Portanto, no modo autônomo, DORIS será controlada por posição, havendo a necessidade de especificar velocidades máximas para cada trecho do trilho. O mapa do trilho é uma matriz de duas colunas: a primeira coluna representa o tipo de trecho do trilho (reta, transição reta-subida, transição reta-descida, subida, descida, curva à direita, curva à esquerda); a segunda coluna se refere ao comprimento do trecho. O robô pode acelerar nas retas, mas deve reduzir a velocidade em curvas, onde há compensação de velocidades devido ao comprimento do robô e à distância entre as rodas em um mesmo gimbal, e em subidas e descidas, onde grandes velocidades produzirão muito esforço e desgaste nos motores/engrenagens.

Durante uma missão do tipo *GOTO*, a tarefa *Motion_Planning_Position*, após obter a localização do robô, envia um pedido de planejamento de velocidades ao processo *MasterPlanning*, em uma mensagem com todas as entradas necessárias. *MasterPlanning* tem acesso ao mapa do trilho, disponibilizado pelo Cartógrafo, e possui um dicionário de três possíveis velocidades máximas (V_{max}): “fast”, “normal” e “slow”, mapeadas em 0.6, 0.3 e 0.1 m/s respectivamente.

As velocidades máximas dos trechos do trilho são proporcionais à V_{max} e escolhidas empiricamente: $V_{reto} = V_{max}$; $V_{reto/subida} = 0.7 * V_{max}$; $V_{reto/descida} = 0.7 * V_{max}$; $V_{curva} = 0.85 * V_{max}$; $V_{subida} = 0.5 * V_{max}$; $V_{descida} = 0.5 * V_{max}$. A fim de garantir que o robô chegue em determinado trecho respeitando a velocidade máxima deste, o *MasterPlanning* calcula a posição de troca de velocidade máxima pela equação de Torricelli: $v_f^2 = v_0^2 + 2a\Delta S$. ΔS representa a distância percorrida, ou seja, a posição de troca de velocidades e a é a aceleração/desaceleração máxima da EPOS

$(a = 0.11m/s^2)$.

3.2.3 Implementação da camada Executivo

A camada Executivo elaborada para a DORIS usa SMACH (subseção 2.5.2) como *framework*, e implementa a técnica de função de coordenação competitiva (subseção 2.3.2), a arquitetura de subsunção, onde o módulo básico de comportamento reativo é uma máquina de estados aumentada (*AFSM*), como a figura 2.17. A camada Executivo é essencial para a autonomia do robô, portanto também é o escopo da implementação do autor.

As subseções a seguir detalham a implementação da camada Executivo, identificando como é realizada cada responsabilidade: Sequenciador, Seletor, Gerenciador de recursos, e Monitoramento da execução e recuperação de erros (*Execution monitoring and error recovery*). Como o ambiente de desenvolvimento da camada Funcional é ROS, optou-se pela utilização da camada Executivo SMACH [50] por demonstrar resultados positivos em diversas aplicações, e já ser integrada ao sistema ROS. Foram adicionadas algumas funcionalidades à camada a fim de garantir todas as responsabilidades de uma camada Executivo. A camada não executa algoritmos de alto processamento, como a camada Funcional, logo foi escolhida a linguagem Python para programação.

A figura 3.6 mostra a arquitetura da camada Planejador, destacando suas responsabilidades e suas interconexões entre camadas.

Sequenciador e Seletor

SMACH é um *framework* para projetar máquinas de estados hierárquicas concorrentes. As máquinas de estados de SMACH possuem características bem peculiares não encontradas em máquinas de estados formais, como a *user data*, comentada na subseção 2.5.2. O **Sequenciador** e **Seletor** desenvolvidos para a DORIS utilizarão as capacidades de SMACH e as vantagens de ROS para a modelagem das tarefas do robô.

Como seres humanos, robôs deveriam possuir um conjunto de processos que estão sempre em execução, e outros conjuntos de processos que só entrarão em execução dependendo da missão a ser executada. Por exemplo, em um ser humano,

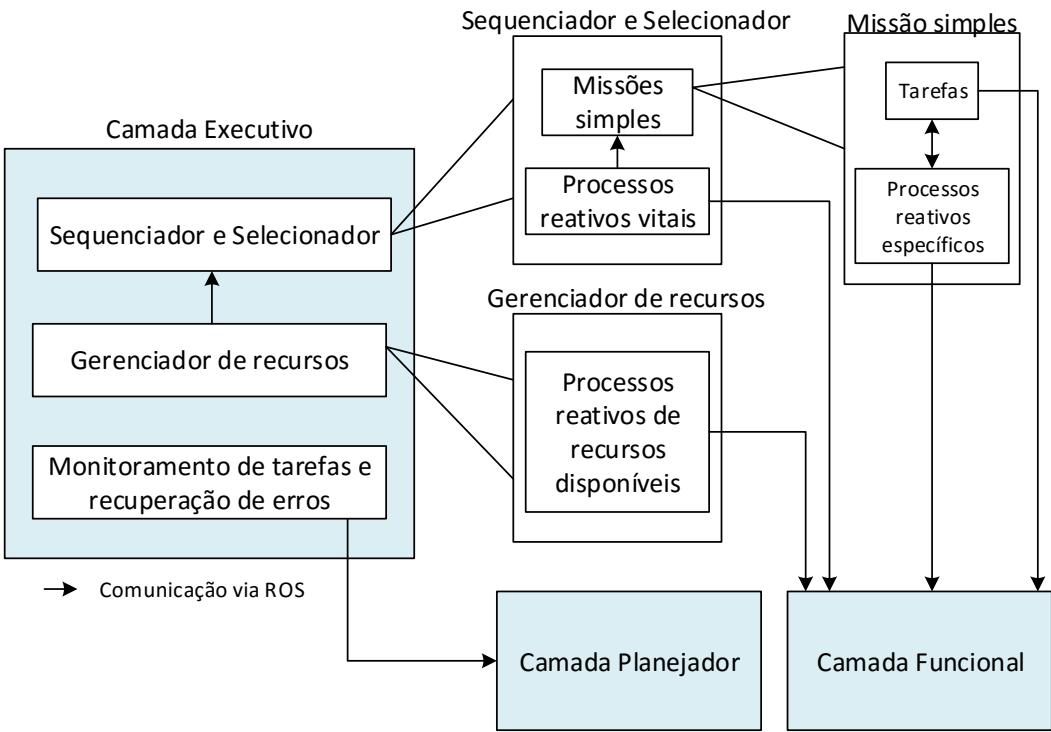


Figura 3.6: Arquitetura da camada Executivo.

os processos vitais, como “respirar” e “o bombeamento do coração”, estão sempre em execução. Alguns podem ser postos em espera por um tempo, como “respirar”, e outros são incontroláveis, como “o bombeamento do coração”.

Definição: **Processo reativo vital** é um processo que deve ser executado sempre que o robô estiver em modo autônomo. É o tipo de processo vital ao robô, garante robustez, proteção ou a sobrevivência ao sistema e aos seus dispositivos.

Dependendo da missão a ser realizada, além das tarefas que a especificam, um conjunto de processos paralelos começa a ser executado de maneira automática a fim de garantir o sucesso da missão, ou como novas formas de proteção do sistema aos novos perigos que a missão pode impor. Para o ser humano, em missões como “palestrar”, “produzir um relatório”, e “ir a um evento”, os processos “falar”, “escrever” e “desviar de pessoas e objetos” são automáticos e executados somente durante a realização das missões.

Definição: **Processo reativo específico** é um processo iniciado paralelamente às tarefas da missão, logo quando esta começa a ser executada. São processos

específicos de uma determinada missão, e inicializados automaticamente. Podem garantir robustez, proteção, sobrevivência do sistema, ou melhorar a execução da missão.

Os **processos reativos vitais** e **processos reativos específicos** podem interromper a execução de tarefas das missões temporaria ou definitivamente. Por exemplo, uma falha no bombeamento do coração cancela imediatamente qualquer missão em execução. Esta característica de supressão dos **processos reativos** foi modelada pela função de coordenação competitiva, subsunção. Desta forma, foi implementada uma classe, derivada de SMACH, para a AFSM padrão de todas as tarefas. A classe também depende de ROS, e foi idealizada para absorver a vantagem do estilo de comunicação *publish/subscriber*, tornando a modelagem das tarefas bastante modular.

O módulo básico que busca modelar a subsunção com novas funcionalidades, chamado **SUPPRESSION_STATE**, é um *container* SMACH do tipo *concurrence* (concorrente), isto é, um *container* composto por dois *containers* que são executados simultaneamente. Um dos *containers* concorrentes é o **MAIN**, composto por tarefas sequenciais que executam a missão. O outro *container* concorrente é o **SUPPRESSION_MAIN**, o qual aborta a missão (*container MAIN*) caso receba uma mensagem ROS 'Abort' em tópico específico e, na sequência, aborta processos dependentes, por exemplo, missões dependentes.

Exemplo do módulo básico

GOTO é uma missão implementada como um *container* **SUPPRESSION_STATE**, composto pelo *container* **MAIN_GOTO**, o qual é uma sequência de tarefas; e pelo *container* **SUPPRESSION_GOTO**, que aborta **MAIN_GOTO** quando uma mensagem ROS 'Abort' é enviada ao tópico *DORIS/MCS/Goto* (figura 3.7).

Como **SUPPRESSION_STATE** é o módulo básico da camada Executivo, todas as missões, tarefas e *processos reativos* são inicializados como *containers* SMACH do tipo **SUPPRESSION_STATE(name, sup_msg, output_keys, input_keys, to_sup)**. *name* é o nome (ou *tag*) da missão ou tarefa e, consequentemente, seu tópico de supressão (tópico que subscreve esperando mensagem para cancelamento) é

DORIS/MCS/name. *sup_msg* é a mensagem que a tarefa espera para ser cancelada, onde o padrão é “Abort”. *output_keys* e *input_keys* são os *user data* SMACH. E *to_sup* são as missões simples que devem ser cancelados caso uma missão complexa seja abortada, por exemplo, caso a missão complexa **INSPECTION** seja abortada, ela deve cancelar as missões **GOTO** e **Detect**.

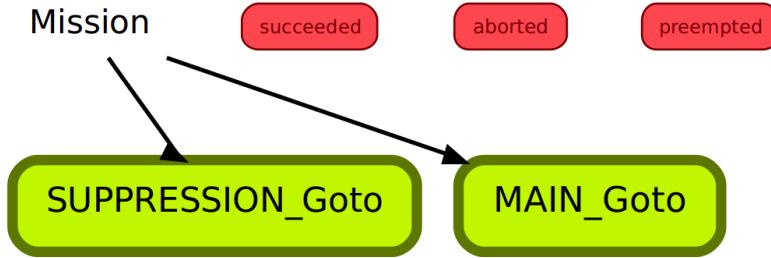


Figura 3.7: Exemplo do módulo básico para a missão simples **GOTO**.

Desta forma, um **processo reativo vital ou específico** pode abortar uma missão, enviando uma mensagem ROS ‘Abort’ ao tópico ‘*DORIS/MCS/MISSION*’. O módulo básico implementado torna a estrutura da camada Executivo muito mais flexível, modular e estimula o paralelismo, sem a preocupação de tarefas conflitantes, já que os conflitos são resolvidos pela subsunção. A subsunção é mais flexível que um esquema de prioridades por IDs, já que o acréscimo de um novo processo exige a alteração de toda uma tabela de prioridades. Além disso, a modelagem do robô fica mais intuitiva e correspondente à natureza.

O problema de *situatedness* (subseção 2.3.2) é resolvido pela decomposição em *processos reativos vitais*, *processos reativos específicos* e pelas tarefas que compõem sequencialmente a missão. Não são todos os comportamentos que estão sempre em execução, mas apenas os essenciais (“vitais”) para o robô e aqueles que possuem alguma relação com a missão (“específicos”). Essa responsabilidade de classificar os processos e escolher qual entrará em ação é chamada de **Selecionador**, foi implementada na camada Executivo e resolve um grande problema de arquiteturas de subsunção. A nível de implementação, as missões, quando são inicializadas, executam os seus *processos reativos específicos* e cancela-os quando são abortadas ou finalizadas.

A responsabilidade **Sequenciador** é quebrar as missões do Planejador em tarefas, que, no caso do SMACH, são sequências de máquinas de estados SMACH.

O programador deve, portanto, modelar a quebra de uma missão em tarefas utilizando o *framework* SMACH e adicionar a decomposição à base de conhecimento da camada Executivo.

Processos reativos vitais na DORIS Os processos reativos vitais criados para a DORIS são:

- **STATE_OF_CHARGE**: obtém os status das baterias se comunicando com o *VSS* da camada Funcional. Caso a carga das baterias esteja inferior a 5%, o robô deve parar todos as missões e processos reativos. O processo envia mensagem 'Abort' aos tópicos de todas as missões: '*DORIS/MCS/INSPECTION*', '*DORIS/MCS/PATROL*', '*DORIS/MCS/RECORD*', '*DORIS/MCS/GOTO*'; e a alguns processos reativos: '*DORIS/MCS/StateOfTempAndHum*'.
- **STATE_OF_TEMPERATURE_AND_HUMIDITY**: obtém as condições de temperatura e umidade externas se comunicando com o *VSS*. Quando as condições estão fora do nível de operação do robô, este suspende as missões. O processo envia mensagem 'Abort' aos tópicos de todas as missões: '*DORIS/MCS/INSPECTION*', '*DORIS/MCS/PATROL*', '*DORIS/MCS/RECORD*', '*DORIS/MCS/GOTO*', e outras.

Processos reativos específicos na DORIS Na DORIS, um **processo reativo específico** foi implementado para a missão simples **GOTO**:

- **OBSTACLE_DETECTION**: detecta a presença de obstáculos. Foram criados dois estados concorrentes SMACH, um faz a detecção por câmera fixa por um algoritmo de anomalias no trilho (figura 3.8), outro pelo sensor de consumo de corrente dos motores (comunicação com EPOSNodelet da camada Funcional, figura 3.9), o qual se muito elevado, representa choque com algum obstáculo (a mesma logica deste estado pode ser encontrado em [68]). Caso um dos sensores detecte obstáculo, o estado seguinte do **OBSTACLE_DETECTION** aborta a tarefa **MOTION_POSITION_CONTROL**, pertencente à missão **GOTO** (mensagem ROS 'Abort' ao tópico '*DORIS/MCS/MOTION-POSITION_CONTROL*'), e controla a velocidade do robô para zero. A figura 3.10 mostra o processo em alto nível.

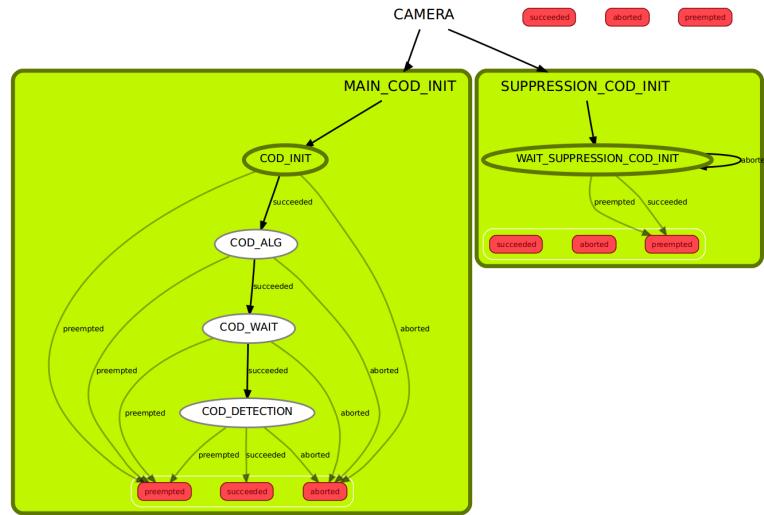


Figura 3.8: Estado **CAMERA** do processo reativo específico **OBSTACLE_DETECTION**.

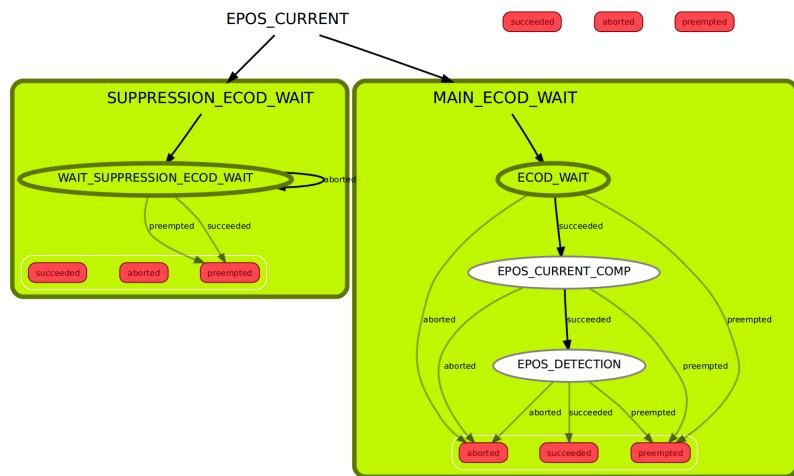


Figura 3.9: Estado **EPOS** do processo reativo específico **OBSTACLE_DETECTION**.

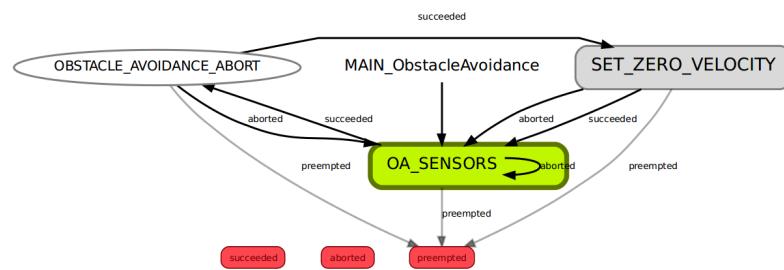


Figura 3.10: Estados do processo reativo específico **OBSTACLE_DETECTION**.

Outro processo reativo específico para a missão **GOTO** é o **AVAILABLE-ENERGY**, ainda não implementado. Este processo poderá calcular a variação da energia consumida durante uma missão e inferir se há energia suficiente para a conclusão. Caso não haja, a missão deve ser abortada.

Tarefas na DORIS As tarefas criadas para a DORIS são detalhadas sequencialmente, conforme a missão simples:

- **GOTO: Localization, PathPlanning, Motion_Control_Position.** Figura 3.11.
 - **Localization:** espera receber mensagem da camada Funcional com a localização e a probabilidade. Caso esta probabilidade seja menor que 50%, o estado **Wander** é ativado. Figura 3.12
 - * **Wander:** estado loop que recebe a localização do robô, posição e probabilidade. Caso a probabilidade seja menor que 50%, o estado se comunica com o *MotionControllerNodelet* e realiza controle de velocidade do robô a $0.1m/s$, até que o robô consiga se localizar com maior precisão. Quando a probabilidade mínima é alcançada, o estado sai com “sucesso”.
 - **PathPlanning:** comunica-se com o *MasterPlanning* da camada funcional para obter o plano de velocidades do robô, no trilho. Seu *user data* é o plano de velocidades.
 - **Motion_Position_Control:** comunica-se com o *PositionControlNodelet* da camada Funcional para enviar o plano de velocidades. A tarefa fica em loop, obtendo localização do robô e enviando o plano de velocidade, conforme a mudança de localização. Figura 3.13.
- **RECORD:** comunica-se com os a camada Funcional para gravação dos dados no SSD embarcado. Espera o comando de ‘Abort’, que será enviado pela missão **STOP_RECORD**. Figura 3.14.
- **STOP_RECORD:** envia mensagem ‘Abort’ ao tópico ‘/DORIS/MCS/RECORD’.

- **DETECT**: comunica-se com os algoritmos de inspeção da camada Funcional. Espera o comando de 'Abort', que será enviado pela missão **STOP_DETECT**.

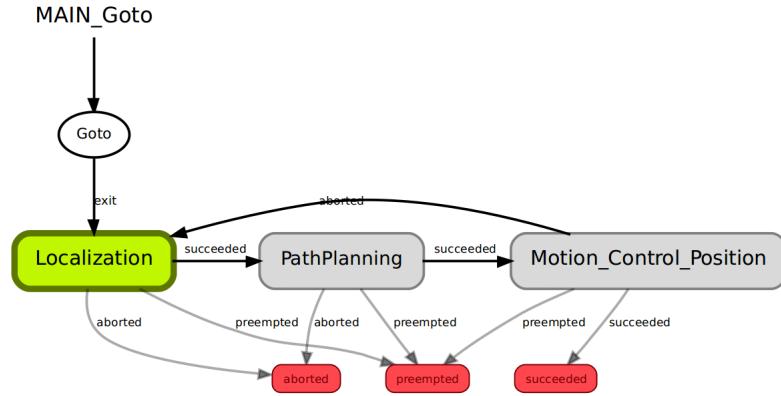


Figura 3.11: Tarefas da missão simples **GOTO**.

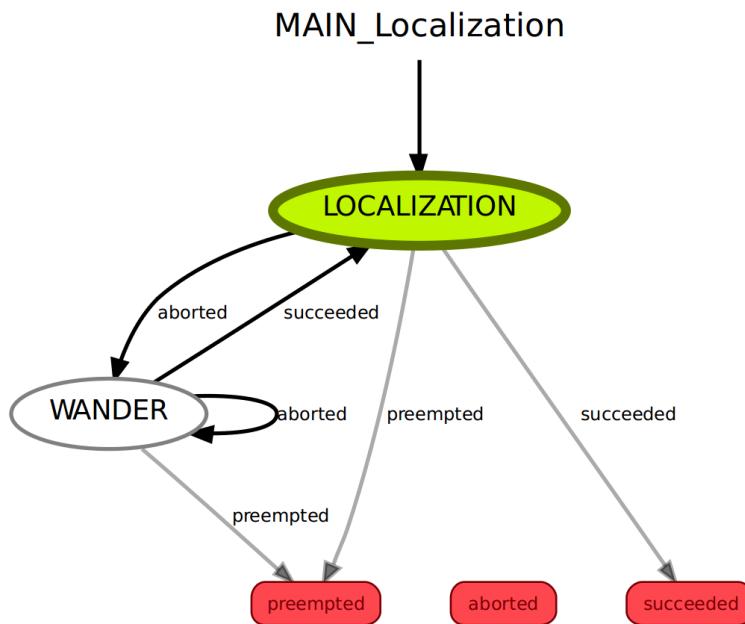


Figura 3.12: Estados da tarefa **Localization**.

Gerenciador de recursos

A camada Executivo tem a responsabilidade de gerenciar os recursos do robô pelas missões, isto é, compartilhar, estabelecer prioridades e verificar a disponibilidade dos sensores e atuadores do robô aos processos em execução.

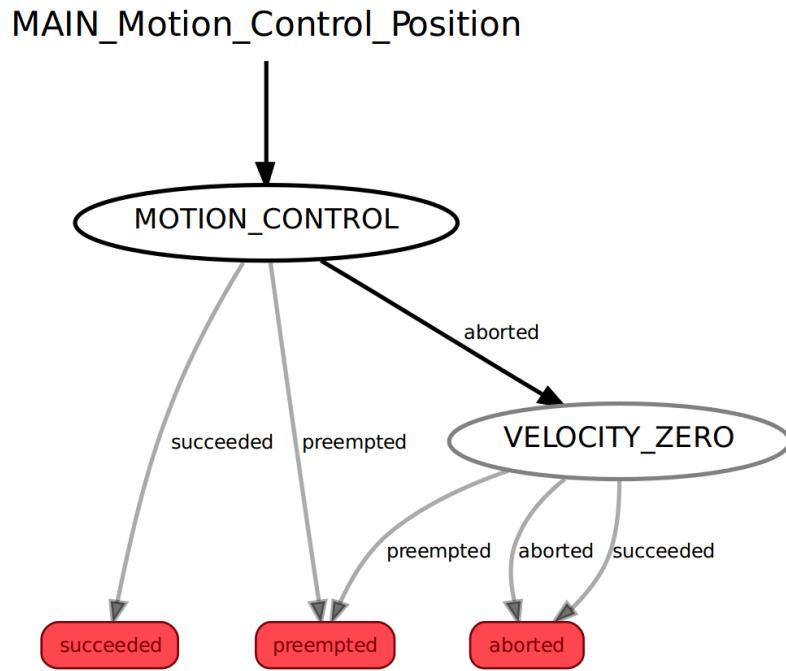


Figura 3.13: Estados da tarefa **Motion_Position_Control**.

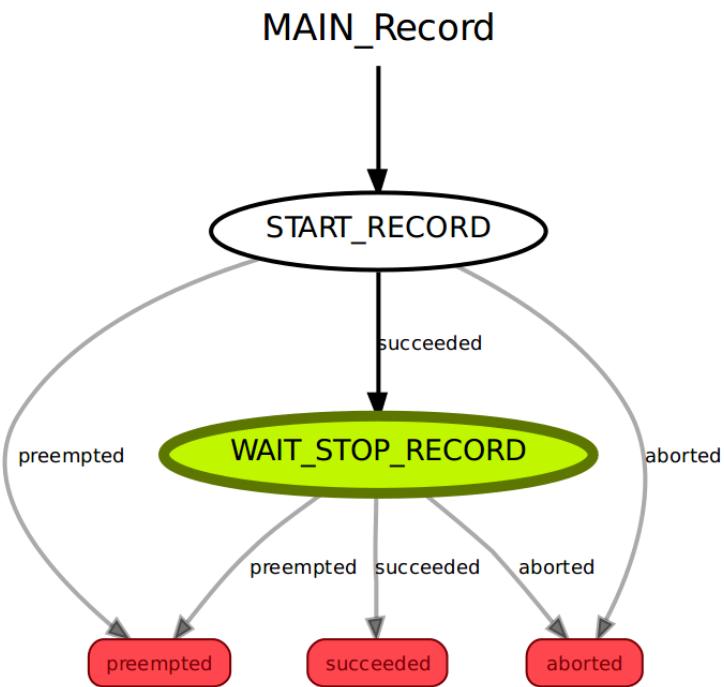


Figura 3.14: Estados da tarefa **RECORD_TYPE**.

O compartilhamento de sensores, dados e processamento de sensores estão sempre disponíveis a todas funções e processos do robô, graças ao estilo de comunicação *publish/subscriber* do *framework* ROS. Os atuadores, no entanto, só estão dis-

poníveis a executar uma tarefa por vez. Por exemplo, o manipulador da DORIS não tem condições de inspecionar dois equipamentos ao mesmo tempo; um robô com um manipulador que serve bebidas não é capaz de buscar três bebidas simultaneamente; um robô submarino com apenas uma câmera sob um sistema *Pan & Tilt* (atuador) não pode compartilhar a câmera para navegação e observação da cauda simultaneamente.

ROS soluciona o problema de compartilhamento de sensores, e o **Sequenciador**, junto com a subsunção, resolve o problema de prioridades. As missões são sequenciais, não paralelas, logo não há perigo de compartilhamento de atuadores entre missões. O conflito ocorre quando processos reativos, paralelos às missões, necessitam do mesmo recurso da missão. O esquema de função de coordenação competitiva é a solução para os conflitos.

Por fim, a disponibilidade de recursos é essencial para a execução das missões. Caso seja dada uma tarefa de inspeção com manipulador, na DORIS, o robô não deve executar todas as tarefas da missão e, ao fim, verificar que o sensor de vibração não está disponível. Os **processos de recursos disponíveis** são individuais para cada recurso e verificam se o sensor está disponível. Caso o recurso não esteja disponível, a missão que o utiliza é automaticamente abortada (subsunção).

Definição: **Processo reativo de recurso disponível** é um processo reativo que monitora o status de um recurso do robô (sensores e atuadores). Este processo pode abortar missões.

Um exemplo de **processo reativo de recurso disponível**, na DORIS, é o **EPOS_RESOURCE**, o qual aborta a missão **GOTO** caso esta esteja no plano de missão do robô, pois não há como o robô se locomover sem o driver da EPOS.

Os **processos reativos de recursos disponíveis** são como os **processos reativos vitais**, ou seja, estão sempre em execução, desde a inicialização do modo autônomo do robô.

Monitoramento de tarefas e recuperação de erros

O controle de missão da camada Planejador fornece ao usuário os status das missões e os erros, junto com os motivos das falhas. Entretanto, ele apenas interpreta e mostra os dados fornecidos pela camada Executivo, isto é, o controle de missão possui um

thread nó de ROS subscritor do tópico *Warning* e *loginfo*. A camda Executivo envia para este tópico os status das missões que estão sendo executadas.

O SMACH possui um monitoramento de tarefas simplificado, de forma que todos os estados publicam no tópico *loginfo* os eventos ocorridos: transições de estados, transições de *containers*, *outcomes* e *user data*. O *Smach Viewer* (visualizador SMACH) e o controle de missão subscrevem a este tópico e, desta forma, é possível gerar um log detalhado ao usuário.

Além do uso do monitoramento simples do SMACH, foi desenvolvida uma classe derivada para reportar acontecimentos de subsunção de processos, ou seja, quando um *processo reativo* aborta uma tarefa ou missão, o *processo reativo* envia automaticamente uma mensagem ao tópico *Warning*, reportando o motivo do cancelamento. Esta informação é a mais valiosa para o operador, visto que é extremamente importante o usuário saber o motivo detalhado do cancelamento de uma determinada missão. Por exemplo, pode haver diversos motivos para o robô falhar a missão *INSPECTION('VIDEO',30)*, e o usuário, sem saber o motivo, pode tentar executar a missão diversas vezes obtendo sempre uma falha “misteriosa”. A um programador seria ainda necessário o debug do código completamente, e mesmo somente nas tarefas que contém a missão é muito trabalhoso. O monitoramento de tarefas exige, portanto, explicações detalhadas da falha a fim de facilitar o uso do robô.

A recuperação de erros pode ser implementada no SMACH como novas tarefas da missão. Como cada estado SMACH pode ter n *outcomes*, a falha de execução de um processo poderia levar a um estado de recuperação. Um exemplo disso é a implementação do **processo reativo específico OBSTACLE_DETECTION**, o qual cancela a tarefa **MOTION_POSITION_CONTROL**. Em caso de cancelamento, a missão **GOTO** não é abortada por inteiro, mas entra em uma tarefa de recuperação, a qual busca uma nova trajetória para a DORIS (sentido contrário, em caso de trilho circular).

Capítulo 4

Resultados e Discussões

O capítulo 3 detalhou a implementação das camadas da arquitetura robótica híbrida de três camadas. O detalhamento da implementação mostrou o caráter modular da arquitetura, o que torna possível o desenvolvimento da solução em estágios independentes de programação. A integração das camadas é trivial pelo *framework* ROS e seu estilo simples de comunicação entre os componenentes de software.

A modularidade da arquitetura permite a avaliação independente de cada camada e, por fim, a integração é apenas um teste de comunicação entre as camadas. Os testes da camada Funcional, no entanto, requerem o hardware, isto é, o robô DORIS. Os testes da camada Executivo e Planejador podem ser simuladas em ambiente de programação. Na seção 4.1 deste capítulo, será desenvolvida uma metodologia para a avaliação da arquitetura híbrida de três camadas e as seções seguintes, 4.2, 4.3, 4.4, resumem os testes de cada camada.

4.1 Metodologia para avaliação das camadas

Há diversos critérios de avaliação de uma arquitetura robótica. De acordo com Arkin [5], podemos avaliar arquiteturas quanto a:

- **Suporte a paralelismo.**
- ***Hardware targetability*:** este conceito se refere a quão bem uma arquitetura pode ser mapeada em sistemas robóticos reais, isto é, sensores e atuadores físicos; e o desempenho computacional. Este critério é exclusivo da camada Funcional.

- **Niche targetability:** quão bem uma arquitetura é capaz de fazer o robô se adaptar ao seu ambiente de operação.
- **Suporte a modularidade:** desde a facilidade de encapsulamento de comportamentos abstratos e componentes (baixo nível) à possibilidade de reutilização da arquitetura para outros robôs (alto nível).
- **Robustez:** em caso de falha de hardwares (sensores, atuadores e etc), a arquitetura deve ser capaz de se recuperar. Quais os mecanismos que a arquitetura possui para contornar falhas?
- **Tempo de desenvolvimento:** quais as ferramentas e *frameworks* disponíveis na arquitetura.
- **Flexibilidade em tempo de execução:** como o sistema de controle pode ser ajustado ou reconfigurado em tempo de execução.
- **Desempenho em executar tarefas.**

A arquitetura híbrida de três camadas foi idealizada para passar com ótima avaliação em todos os critérios de Arkin. Entretanto, como já explicitado na subseção 2.4.2, há diversas formas de implementar esta arquitetura e esta liberdade de programação acaba por não garantir boa avaliação nos critérios estabelecidos. É importante que as camadas sejam projetadas a cumprirem os critérios de forma satisfatória.

As camadas desenvolvidas para a DORIS foram projetadas para passar satisfatoriamente nos critérios de Arkin e de outros roboticistas. Dessa forma, a metodologia de avaliação e testes é verificar o desempenho nos critérios e destacar as responsabilidades implementadas em cada camada e como elas cumprem os requisitos.

As camadas Executivo e Planejador compõem o sistema autônomo do robô. Por mais robusto que o sistema seja, e mesmo que haja possibilidade de suspender ações em tempo real, as camadas de nível superior devem ser exaustivamente simuladas em ambientes de programação e em computador semelhante ao embarcado no robô. Como a camada Funcional é a camada que faz interface com os hardwares do robô (sensores e atuadores), simulações não bastam para a avaliação dos critérios de Arkin, logo são necessários testes exaustivos, em campo, com o robô.

4.2 Testes da implementação da camada Funcional

A camada funcional é a primeira a ser implementada na camada da arquitetura híbrida de três camadas. Somente com a camada funcional é possível enviar comandos aos atuadores, ou seja, controlar o robô por *joystick* ou com uma interface simples de usuário. É possível observar os dados dos sensores, processar os dados e testar funcionalidades básicas do robô.

A camada Funcional desenvolvida para DORIS é detalhada na subseção 3.2.2 e, como já foi analisada, segue o modelo “ideal” de implementação de uma camada Funcional descrita por Quigley [41], desenvolvedor do ROS, e Volpe [34], desenvolvedor do CLARAty. Há pacotes genéricos a serem atribuídos a qualquer robô, pacotes específicos da DORIS e pacotes com funcionalidades ROS.

Com o auxílio das *tools* (subseção 3.2.2), componentes gráficos para o operador enviar comandos ao robô, o usuário pode teleoperar o robô e observar as saídas dos sensores. A figura 4.1 mostra a teleoperação do SAM (*Single Autonomous System*), predecessor da DORIS, em uma interface web. Na figura, pode-se observar a imagem enviada pela câmera interna do robô e o ambiente 3D em que o robô está inserido, em RVIZ. Na figura 4.2, mostram-se a interface de controle da DORIS com dados de corrente e velocidade dos motores, a saída de vídeo da câmera interna ao robô, e a saída de vídeo de uma câmera externa ao robô.

A camada Funcional desenvolvida no *framework* ROS mostrou ***Hardware targetability***. Além de ROS possuir um grande repositório de drivers (interface hardware-software), os componentes específicos desenvolvidos apresentaram ótimo desempenho.

O *framework* ROS é uma ferramenta que, usada de maneira correta, faz com que a camada Funcional passe por quase todos os critérios de Arkin. Podemos avaliar a camada Funcional com os critérios estabelecidos:

- **Supporte a paralelismo:** é inerente a sistemas *multi-thread* (computadores embarcados com sistemas operacionais que permitem o paralelismo), e aos estilos de comunicação *publish-subscriber* e *service*, que permitem que os componentes obtenham, ao mesmo tempo, acesso aos diversos dados de sensores

do robô.

- **Hardware targetability:** os diversos robôs que utilizam ROS já mostram por si só o **Hardware targetability** desta ferramenta.
- **Niche targetability:** ROS também já foi utilizado por uma variedade de aplicações robóticas. Além disso, os resultados em DORIS mostram que o código implementado pode ser estendido a robôs com desafios semelhantes.
- **Suporte à modularidade:** os componentes e pacotes desenvolvidos são modulares e seu nível de abstração permite a utilização em outros robôs. O esquema de classes e herança é essencial para alcançar este objetivo.
- **Robustez:** em caso de falha, a camada não entra em colapso e mensagens de falha são enviadas a camadas superiores (Executivo e Planejador). Mas a camada Funcional não é projetada para recuperação de falhas, responsabilidade de camadas superiores.
- **Tempo de desenvolvimento:** o grande repositório disponível para o *framework* ROS reduz muito o tempo de desenvolvimento. Além disso, uma metodologia de programação, e a separação em módulos facilita o desenvolvimento de sistemas robóticos.
- **Flexibilidade em tempo de execução:** os resultados de DORIS com a teleoperação mostrou que pode ser controlada com alteração dos parâmetros de controle em tempo real. O sistema de comunicação de ROS permite a implementação de *tools* que se comunicam com os componentes do robô (interação base-robô) e cria esta flexibilidade em tempo de execução.
- **Desempenho em executar tarefas:** os resultados bem sucedidos mostraram o alto desempenho da DORIS.

A camada Funcional permite, por si só, a teleoperação do robô e testes com o sistema. Apesar de passar com excelência pelos critérios de Arkin, a camada não transforma DORIS em um sistema autônomo. As outras camadas, além de garantirem esta nova configuração do sistema, devem também cumprir os critérios de Arkin.

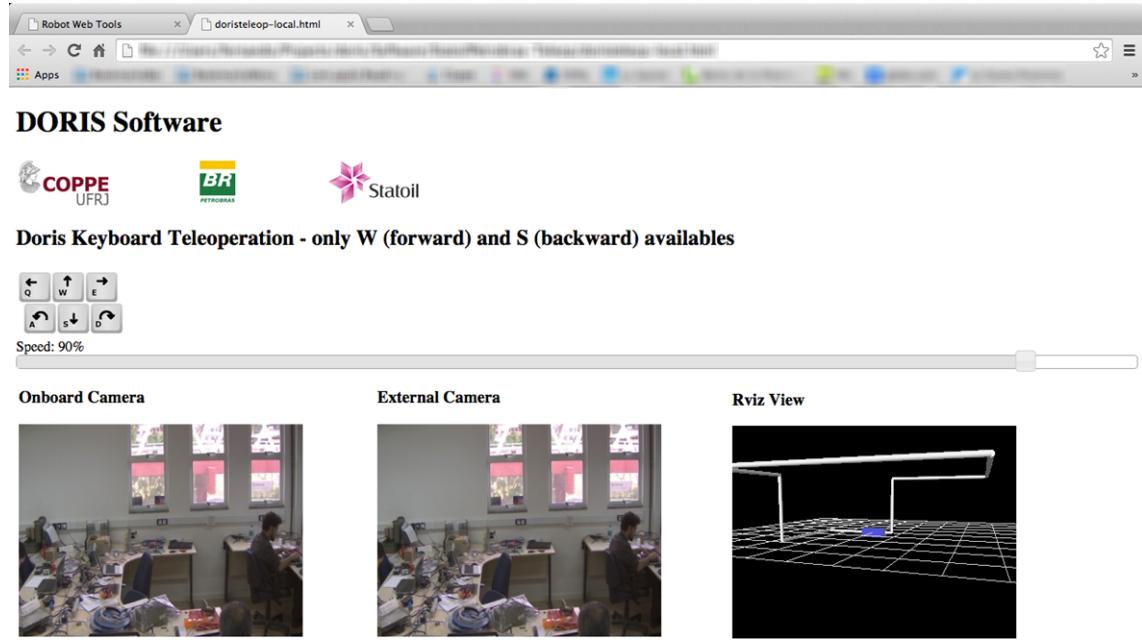


Figura 4.1: Teleoperação da DORIS.

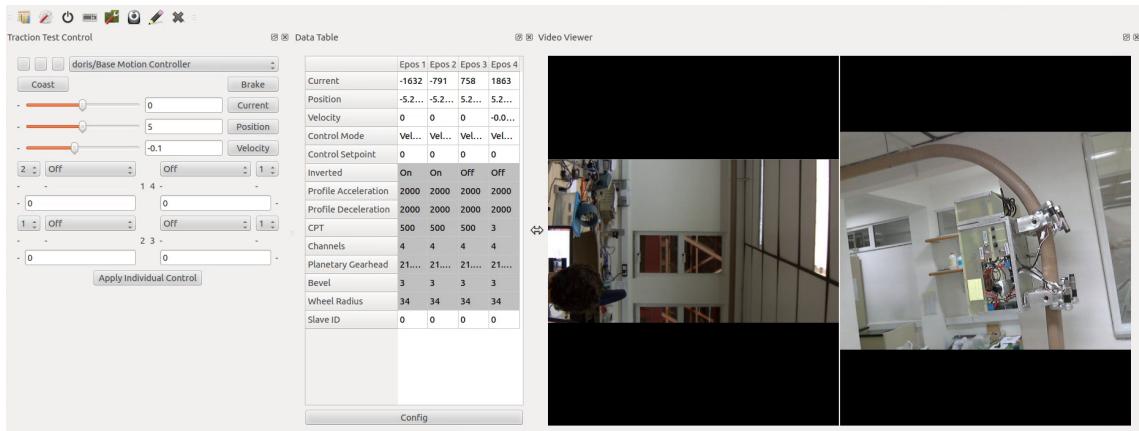


Figura 4.2: Interface de controle da DORIS.

4.2.1 Testes de planejamento de velocidades

Apesar de pertencer à camada Funcional da arquitetura proposta, o planejamento de velocidades é um *thread* que pertence ao desenvolvimento de um sistema autônomo. O planejamento de velocidades é um algoritmo cuja saída são as velocidades para cada trecho do trilho, ele não se comunica com outros elementos da camada Funcional, não é interface ou driver, mas comunica-se com componentes das camadas

Executivo e Planejador. Dessa forma, como as camadas de nível superior, o algoritmo é simulado, em vez de ser testado exaustivamente no robô.

Como já documentado na subseção 3.2.2, a função **MasterPlanning** possui os argumentos: localização atual do robô; localização objetivo; direção de movimento (1 ou -1 - sentido natural ou sentido contrário); número de voltas desejado antes de chegar à localização objetivo; e se o trilho é circular (verdadeiro ou falso). A saída do algoritmo é uma matriz com n linhas e três colunas: a primeira coluna representa a posição do trilho onde ocorre troca de velocidade máxima; a segunda coluna é a velocidade máxima; a terceira é o *set-point* de posição.

Exemplo

Suponha que o robô está na posição 0 do trilho e começa a execução da tarefa *Motion_Planning_Position* da missão *GOTO(10, 'fast')* (ver subseção 3.2.1). A função *master_planning(0, 'fast', 10)* retorna a matriz 4.1, que pode ser interpretada graficamente em 4.3.

Tabela 4.1: Matriz de velocidades máximas em cada trecho do trilho do ponto 0 ao 10.

Ponto de troca (m)	V_{max} (m/s)	Set-point (m)
0	0.6	10
1.907	0.42	10
3.355	0.3	10
4.495	0.42	10
5.506	0.6	10
5.847	0.51	10
7.308	0.6	10
8.334	0.42	10
9.282	0.3	10

O mapa das seções do trilho até a posição 10, disponível pelo Cartógrafo, está representado na tabela 4.2. Observe que o trecho inicial “Reto” possui comprimento 2.733 m e velocidade máxima 0.6 m/s, mas antes de o robô chegar no final do trecho, ele começa o processo de desaceleração em 1.907 m de forma a ser possível fazer a troca de trecho na velocidade máxima permitida para a “Transição reto-subida”

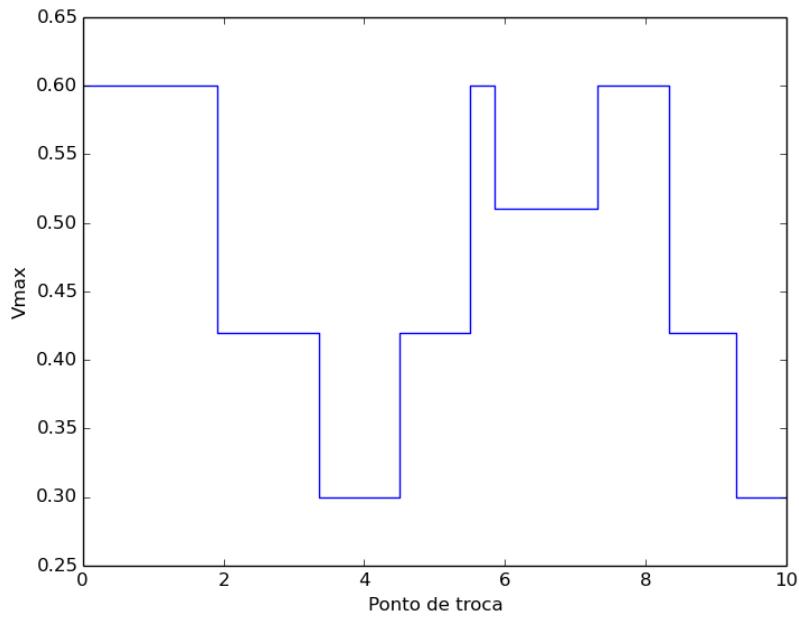


Figura 4.3: representação gráfica do plano de velocidades do robô para uma missão de GOTO(10,’fast’).

(0.42 m/s).

Tabela 4.2: Mapa das seções do trilho

Tipo de trecho	Comprimento (m)	Soma (m)
Reto	2.733	2.733
Transição reto-subida	1.011	3.744
Subida	0.750	4.495
Transição subida-reto	1.011	5.506
Reto	0.790	6.297
Curva à esquerda	1.011	7.308
Reto	1.852	9.160
Transição reto-descida	0.510	9.670
Descida	0.790	10.46

Caso seja inserida uma posição alvo maior que o comprimento do trilho, e o trilho for circular, considera-se uma volta e tira-se a diferença, por exemplo se o trilho posui comprimento 140 e é inserido posição alvo 200, considera-se uma volta e posição alvo 60.

4.3 Testes da implementação da camada Executivo

A avaliação da camada Executivo é a análise e testes por simulação de suas responsabilidades dentro dos critérios de Arkin. Isto é, como o modo de implementação de cada responsabilidade da camada se comporta em face aos critérios estabelecidos.

A criação de diversos conceitos de processos que são executados simultaneamente, como **processos reativos vitais, específicos e de recursos disponíveis**, a decomposição de missões em tarefas, e a possibilidade de tarefas concorrentes (*concurrency*), são métodos para criação de uma camada com **Suporte a paralelismo** e **Suporte à modularidade**. Além disso, os conceitos criam um método intuitivo para a implementação da camada, seguindo o modelo da natureza do homem e outros animais, levando a um menor **Tempo de desenvolvimento**. No entanto, paralelismo e modularidade podem gerar conflitos, os quais devem ser resolvidos por uma função de coordenação. A arquitetura híbrida proposta utiliza a subsunção como função de coordenação, e são necessárias avaliações e testes para verificar sua eficiência.

A simulação consiste na implementação dos **processos reativos**, tarefas e missões simples da DORIS, utilizando a classe implementada **SUPPRESSION-STATE**, apresentada na subseção 3.2.3, e no envio de mensagens ROS aos componentes da camada, simulando mensagens do nível Funcional (saídas de sensores e atuadores do robô). Como já foi discutido, a classe derivada de SMACH é o módulo comportamental da camada Executivo e faz o papel da função de coordenação por subsunção. Os **processos reativos** e as missões da DORIS já foram documentados na subseção 3.2.3 e 3.2.1, respectivamente.

A metodologia de simulação é composta por três estágios: 1) Para cada tarefa de missão simples, simular as possibilidades de *outcomes* (resultados das tarefas), isto é, as ramificações de cada tarefa; 2) Simular as interações com os três tipos de **processos reativos**; e 3) Verificar a finalização da missão, quando todos as tarefas são completadas. Abaixo, é demonstrada a simulação de uma missão simples. O visualizador *smach_viewer* é utilizado durante as etapas de simulação, por disponibilizar os dados dos estados SMACH em tempo real.

Exemplo - simulação missão simples GOTO(80,’fast’)

Ao inicializar o sistema autônomo com a missão simples **GOTO(80,’fast’)** (robô deve se locomover até a posição 80 do trilho, em modo de velocidade rápido e pelo menor caminho possível), os **processos reativos** são executados em paralelo automaticamente. A figura 4.4 é uma imagem do visualizador *smach_viewer* que ilustra os processos em execução: em vermelho está destacada a missão simples **GOTO**; em azul, o **processo reativo de recurso disponível: EPOS**, o qual verifica o status dos motores e drivers EPOS (hardwares); em verde, os **processos reativos vitais: Charge**, que verifica o status da bateria, e **StateOfTemp**, que verifica a temperatura e umidade do robô; e em amarelo, o **processo reativo específico: OA (ObstacleAvoidance)**, que detecta objetos no trilho.

A tarefa inicial da missão simples **GOTO** é **Localization**, na qual permanece até receber o valor da posição do robô e sua probabilidade. Caso esta probabilidade esteja dentro do esperado (entre 0.6 e 1), o robô segue para a tarefa **PathPlanning**. A tarefa **PathPlanning** requisita o plano de velocidades ao algoritmo **MasterPlanning** da camada Funcional. Quando recebido, a missão **GOTO** segue para sua última tarefa **Motion_Control_Position**. Esta é um loop que recebe a posição do robô, compara com a tabela recebida do **PathPlanning** e envia comando de controle de posição ao componente **PositionController** da camada Funcional (figura 3.11).

Como a primeira tarefa **Localization** espera mensagem de ROS da camada Funcional, uma mensagem ROS com posição e probabilidade é enviada pelo terminal (Ubuntu) ao tópico *DORIS/Vehicle/Localization*, simulando a camada de baixo nível. Ao receber a mensagem '[30,1]' (posição 30 e probabilidade 1), a tarefa é completada, e a tarefa seguinte, **PathPlanning**, recebe o dado de posição por *user data* SMACH. **PathPlanning** envia por mensagem de ROS comandos ao **MasterPlanning**, recebe o plano deste e o envia à tarefa **Motion_Control_Position** também por *user data* SMACH.

A figura 4.5 mostra as transições das tarefas descritas pelo terminal do Ubuntu onde: em azul, o recebimento da localização e dados da missão (30 é posição atual, 80 é posição objetivo, ’fast’ é o modo de velocidade rápida); em verde, a matriz de velocidades e pontos de troca gerados pelo **MasterPlanning** da camada Funcio-



Figura 4.4: Processos em execução durante missão **GOTO**.

nal e recebida pela tarefa **PathPlanning** da missão **GOTO** (camada Executivo); e, em vermelho, o loop da tarefa **Motion_Control_Position**, a qual recebe uma localização e controla o robô por posição. Na figura 4.6, a mesma transição pode ser vista no smach_viewer, no qual, em verde, são as tarefas em execução (no caso, apenas o **Motion_Control_Position**, e o estado de supressão).

De acordo com a metodologia de simulação, para cada tarefa, devem ser testadas os possíveis *outcomes*. As tarefas **Localization** e **Motion_Control_Position** possuem ramificação dependentes de dados da camada Funcional: quando **Localization** recebe dados de posição do robô com certeza inferior a 60%, seu *outcome* é o estado **WANDER**, o qual controla o robô com velocidade 0.1 m/s até a certeza de posição aumentar para 60%; quando **Motion_Control_Position** recebe dados de posição com certeza inferior a 60%, seu *outcome* é **Localization**. Ambas as ramificações são testadas por mensagens de ROS via terminal, por exemplo enviando a mensagem '[30,0.4]' (probabilidade 0.4) ao tópico *DORIS/Vehicle/Localization*, e suas transições são acompanhadas pelo smach_viewer.

```
[WARN] [WallTime: 1454913048.450054] Starting planning...
[INFO] [WallTime: 1454913048.450156] Flag:[ON', 30, 'fast', 80, 1, 0, True]
[INFO] [WallTime: 1454913048.450555] State machine transitioning 'PATH_PLANNING'
:'succeeded'-->'PATH_PLANNING_CALC'
[INFO] [WallTime: 1454913048.451476] Plan: [[28.77254287679999, 0.6, 80], [39.07
5273672267613, 0.51, 80], [40.5358476368, 0.6, 80], [52.34415746518372, 0.42, 80
], [53.792584664274692, 0.3, 80], [54.771949227199997, 0.42, 80], [55.7830928703
99997, 0.6, 80], [72.086486815583726, 0.42, 80], [73.534914014674698, 0.3, 80],
[74.514278577599995, 0.42, 80], [75.525422220799996, 0.6, 80], [79.9515062978676
1, 0.51, 80]]
[INFO] [WallTime: 1454913048.451761] waiting for plan...
[INFO] [WallTime: 1454913048.552603] Plan sent
[INFO] [WallTime: 1454913048.652864] Got message.
[INFO] [WallTime: 1454913048.653402] State machine terminating 'PATH_PLANNING_CA
LC': 'succeeded':'succeeded'
[INFO] [WallTime: 1454913048.653736] Concurrent state 'MAIN_Path_Planning' retur
ned outcome 'succeeded' on termination.
[INFO] [WallTime: 1454913048.750460] waitForMsg is preempted!
[INFO] [WallTime: 1454913048.750791] State machine terminating 'WAIT_SUPPRESSION
_Path_Planning': 'preempted':'preempted'
[INFO] [WallTime: 1454913048.751077] Concurrent state 'SUPPRESSION_Path_Planning
' returned outcome 'preempted' on termination.
[INFO] [WallTime: 1454913048.754513] Concurrent Outcomes: {'SUPPRESSION_Path_Pl
anning': 'preempted', 'MAIN_Path_Planning': 'succeeded'}
[INFO] [WallTime: 1454913048.754806] State machine transitioning 'PathPlanning':
'succeeded'-->'Motion_Control_Position'
[INFO] [WallTime: 1454913048.755272] Concurrence starting with userdata:
['in_PathPlanning']
[INFO] [WallTime: 1454913048.756375] State machine starting in initial state 'MO
TION_CONTROL' with userdata:
['in_PathPlanning']
[INFO] [WallTime: 1454913048.756479] State machine starting in initial state 'WA
IT_SUPPRESSION_Motion_Control_Position' with userdata:
[]
[INFO] [WallTime: 1454913048.756932] Waiting localization...
[INFO] [WallTime: 1454913048.757171] waiting suppression of Motion_Control_Pos
ition...
[INFO] [WallTime: 1454913048.757410] Got message.
[INFO] [WallTime: 1454913048.759037] Actual control is (actual position, Vmax, p
osition to go):[28.7725428768, 0.6, 80]
[INFO] [WallTime: 1454913048.759274] Waiting localization...
```

Figura 4.5: Transições da missão simples **GOTO** no terminal.

A segunda etapa da simulação são as três possíveis interações entre os tipos de processos reativos. Devem ser observadas as características de subsunção: cancelamento de missão por processo reativo vital; cancelamento de missão por processo reativo de recurso disponível; e interrupção de missão e recuperação de falha por processo reativo específico.

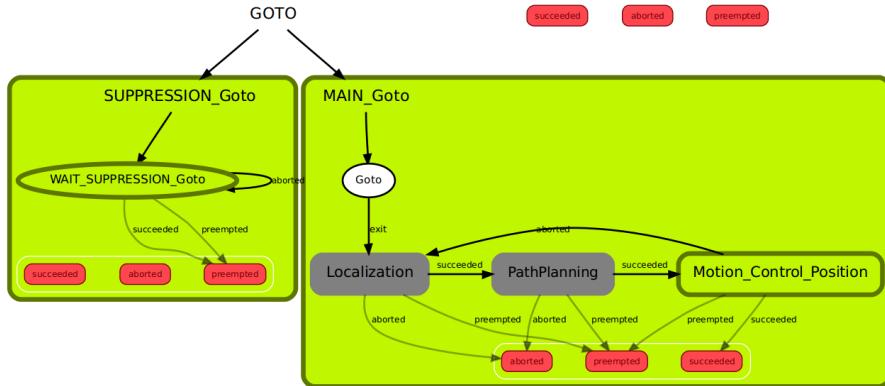


Figura 4.6: Transições da missão simples **GOTO** no smach_viewer.

Exemplo com processo reativo vital: o processo **Charge** recebe a mensagem “5” no tópico *’DORIS/MCS/StateOfCharge/Status’* (5% de nível de bateria) e aborta a missão **GOTO** por segurança, figura 4.7 (terminal) e figura 4.8 (smach_viewer, tarefa em cinza significa que não está em execução).

```
[INFO] [WallTime: 1454916961.300553] Got message.
[INFO] [WallTime: 1454916961.300961] State machine transitioning 'STATE_CHARGE_COMPARE'
:'succeeded'-->'STATE_CHARGE_ABORT'
[WARN] [WallTime: 1454916961.301458] Trying to abort Goto states due to charge state
```

Figura 4.7: Ao receber uma informação de nível de bateria inferior a 5%, **Charge** aborta a missão **GOTO** (terminal).

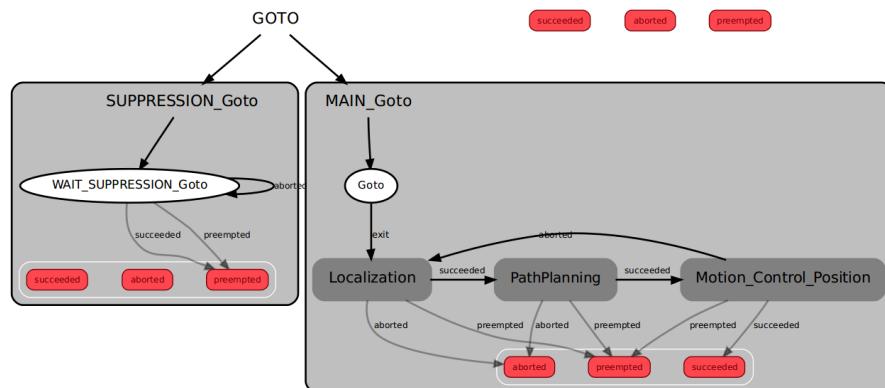


Figura 4.8: Ao receber uma informação de nível de bateria inferior a 5%, **Charge** aborta a missão **GOTO** (smach_viewer).

Exemplo com processo reativo de recurso disponível: o processo **Epos** recebe a mensagem “False” no tópico *’DORIS/MCS/EPOS>Status’* (recurso não disponível)

e aborta a missão **GOTO**, figura 4.9 (terminal).

```
[INFO] [WallTime: 1454918145.768202] Got message.  
[INFO] [WallTime: 1454918145.768647] State machine transitioning 'EPOS_STATUS_COMPARE':'su  
cceeded'-->'EPOS_ABORT'  
[WARN] [WallTime: 1454918145.769125] Trying to abort Goto states due to EPOS resource not  
found
```

Figura 4.9: Ao receber uma informação de recurso indisponível, **Epos** aborta a missão **GOTO** (terminal).

Exemplo com processo reativo específico: o processo **OA (ObstacleAvoidance)** recebe a mensagem “8” no tópico *’DORIS/Vehicle/EPOS/Current’* (corrente consumida do motor maior que 8 ampères) e aborta a tarefa **Motion_Control_Position** da missão, para um estado de recuperação. A missão volta à tarefa **Localização**, o cartógrafo deve atualizar o mapa do trilho com o obstáculo para recálculo da trajetória (sentido contrário).

Dessa forma, as responsabilidades **Sequenciador** e **Selecionador** da camada Executivo atendem aos critérios **Suporte a paralelismo**, **Suporte à modularidade** e **Tempo de desenvolvimento**, utilizando a metodologia de simulação adotada. Além disso, o **Desempenho em executar tarefas** é aumentado com o paralelismo e a modelagem seguindo a metodologia e conceitos de processos estabelecidos. A **Robustez** é alcançada pelas ramificações das tarefas, pelos processos reativos, e está contida na responsabilidade de recuperação de falhas. A modelagem das tarefas não pode ser realizada em tempo de execução, o que não é desvantagem, pois a metodologia de simulação deve ser seguida e sistemas autônomos não devem ser executados sem testes prévios. A **Flexibilidade em tempo de execução** deve estar disponível na camada Funcional, mas não na camada autônoma.

Por fim, o **Niche targetability** é muito abrangente, já que a camada comporta robôs modelados por tarefas sequenciais e processos reativos paralelos, o mecanismo mais comum encontrado na natureza.

4.4 Testes da implementação da camada Planejador

Assim como a camada Executivo, os testes da camada Planejador é a avaliação das responsabilidades desenvolvidas para a camada no contexto dos critérios de Arkin,

ou seja, é a análise de **controle de missão**, **agendador** e **cartógrafo** em face aos critérios.

As definições de tipos de missões, simples, complexas e desconhecidas, introduzidas na subseção 3.2.1, mostram a decomposição estabelecida, e estimulam uma implementação em módulos na camada Planejador. O **controle de missão** traduz as missões complexas, isto é, decompõe as missões complexas em missões simples, de maneira sequencial ou paralela, provendo o **Suporte à modularidade** e **Suporte ao paralelismo**.

Na classe missão, três métodos devem ser implementados: *mission(arguments)*, onde as tarefas da missão são implementadas sequencialmente, pertencente à camada Executivo; *reactives(arguments)*, onde são definidos os *processos reativos específicos* da missão, também pertencente à camada Executivo; e *execute(arguments)*, onde o método *mission(arguments)* é executado, junto com as missões simples que compõe a missão, paralela ou sequencialmente. A organização cria uma ferramenta para implementação de missões, agilizando o **Tempo de desenvolvimento**. Além disso, a flexibilidade na modelagem das missões complexas e o paralelismo permitem a otimização do **Desempenho em executar tarefas**, sem comprometer a camada Executivo.

A **Robustez** da arquitetura pertence à camada Executivo, nas ramificações dos *outcomes*, e aos **Agendador** e **controle de missão**, na camada Planejador: erros nas missões agendadas devem ser reprogramadas para o futuro; e o feedback ao usuário disponível pelo **controle de missão** é uma informação que pode ser interpretada e utilizada para algumas tomadas de decisão.

O **Niche targetability** é garantido pela flexibilidade na implementação das missões complexas do **controle de missão**, e a diversidade do **cartógrafo**, o qual pode gerar diversos modelos de mundo pelos os algoritmos da camada Funcional.

Como a camada Executivo, a **Flexibilidade em tempo de execução** é comprometida propositalmente para simulações serem exaustivamente avaliadas antes da execução do sistema autônomo no robô.

Exemplo - **simulação** **missão** **complexa** **INSPECTION(['VIDEO'],[80,'fast'])**

Na simulação da camada Planejador, é avaliado apenas o **controle de missão**,

pois, apesar de as outras responsabilidades terem sido discutidas previamente e seu funcionamento interno detalhado, elas não foram totalmente implementadas e integradas à arquitetura.

As etapas da simulação são: requisição de missão complexa pelo usuário; tradução da missão complexa; execução da missão; feedback ao usuário. A interface gráfica de usuário não está finalizada, mas as mensagens de usuário podem ser simuladas por mensagens ROS via terminal.

Na camada Planejador, há três *threads* esperando mensagens de ROS da interface de usuário: 1) *thread* que aguarda a mensagem da missão; 2) *thread* que espera o comando “Play”, o qual dá início a execução; e 3) *thread* **STOP**, que finaliza a execução, isto é, aborta o sistema autônomo (todas as suas missões e processos). Para o exemplo de missão complexa **INSPECTION**, são enviadas as mensagens: 1) ‘[[1,[0],’slow’,80,1,0,True]]’ ao tópico *’DORIS/MCS/Mission’* (mensagem de missão complexa, onde 1 representa a missão **INSPECTION**, [0] representa inspeção por vídeo, e [’slow’,80,1,0,True] são os parâmetros da missão simples **GOTO**); 2) “Play” ao tópico *’DORIS/MCS’*, inicializando a execução do sistema autônomo.

O controle de missão chama o método *execute(arguments)* da missão complexa **INSPECTION**, o qual a decompõe, como pode ser visto na figura 4.10. Na figura, temos:

- Em verde, primeiramente, é executada a missão principal **INSPECTION-INIT**, composta por uma tarefa (módulo **SUPPRESSION**), cujo **SUPPRESSION_MAIN** aguarda a finalização da missão, isto é, fim da missão **INSPECTION**. Esta missão principal é necessária, pois ela que “segura” a execução de toda a missão complexa, podendo cancelá-la por completo, se requisitada;
- Em azul, logo em seguida, é executada a missão simples **DETECT**, que inicializa o algoritmo de detecção de anomalias. É uma missão em paralelo, pois ela é executada até o fim da missão complexa;
- Em vermelho, estão representadas as missões simples sequenciais. “join()” em um *thread* significa que uma missão simples está sendo executada em uma nova *thread*, mas a *thread* principal (missão complexa) só prossegue

após a finalização da missão simples, o que mostra o caráter sequencial. As missões simples sequenciais são: **GOTO(80,’slow’)**, **StopDETECT** e **StopINSPECTION**. A missão simples **StopINSPECTION** deve existir, pois finaliza a missão simples principal **INSPECTION_INIT**.

- Como não há processos reativos específicos, não há comando de finalização destes.

```
def execute(*args):
    # START MAIN MISSION

    InspectionContainer, InspectionStop = inspection()
    Inspectionthread = threading.Thread(target=InspectionContainer.execute, args=())
    Inspectionthread.start()

    # THREADS IN PARALLEL

    Detectthread = threading.Thread(target=Detect.execute, args=(args[0]))
    Detectthread.start()

    # THREADS IN SEQUENCE

    Gotothread = threading.Thread(target=Goto.execute, args=(args[1]))
    Gotothread.start()
    Gotothread.join()

    StopDetectthread = threading.Thread(target=StopDetect.execute, args=())
    StopDetectthread.start()
    StopDetectthread.join()

    InspectionStopthread = threading.Thread(target=InspectionStop.execute, args=())
    InspectionStopthread.start()
    StopDetectthread.join()

    # STOP REACTIVES

    return |
```

Figura 4.10: Método *execute* da missão **INSPECTION**.

Essa metodologia deve ser seguida em todas as implementações de missões complexas e, futuramente, em missões desconhecidas. A análise da simulação (execução da missão complexa) se torna, então, equivalente à análise da execução das missões simples que a compõe, e pode ser realizada pelo smach_viewer. A figura 4.11 mostra as missões simples e processos reativos em execução, quando a missão complexa **INSPECTION** é inicializada. Segue a legenda de cores:

- Em vermelho, estão destacadas as missões simples: **GOTO** (sequencial) e **DETECT** (paralela);

- Em azul, o **processo reativo de recurso disponível: EPOS**, o qual verifica o status dos motores e drivers EPOS (hardwares). Observe que há a necessidade de implementação de outro **processo reativo de recurso específico: CAMERA**, que verifica o status da câmera e pode abortar a missão **DETECT** e **INSPECTION** caso o recurso não seja detectado;
- Em verde, os **processos reativos vitais: Charge**, que verifica o status da bateria, e **StateOfTemp**, que verifica a temperatura e umidade do robô;
- Em amarelo, o **processo reativo específico** da missão simples **GOTO: OA** (*ObstacleAvoidance*), que detecta objetos no trilho. Neste exemplo, as outras missões simples não possuem **processos reativos específicos**, caso tivessem, estes seriam executados sequencialmente, juntos às missões.

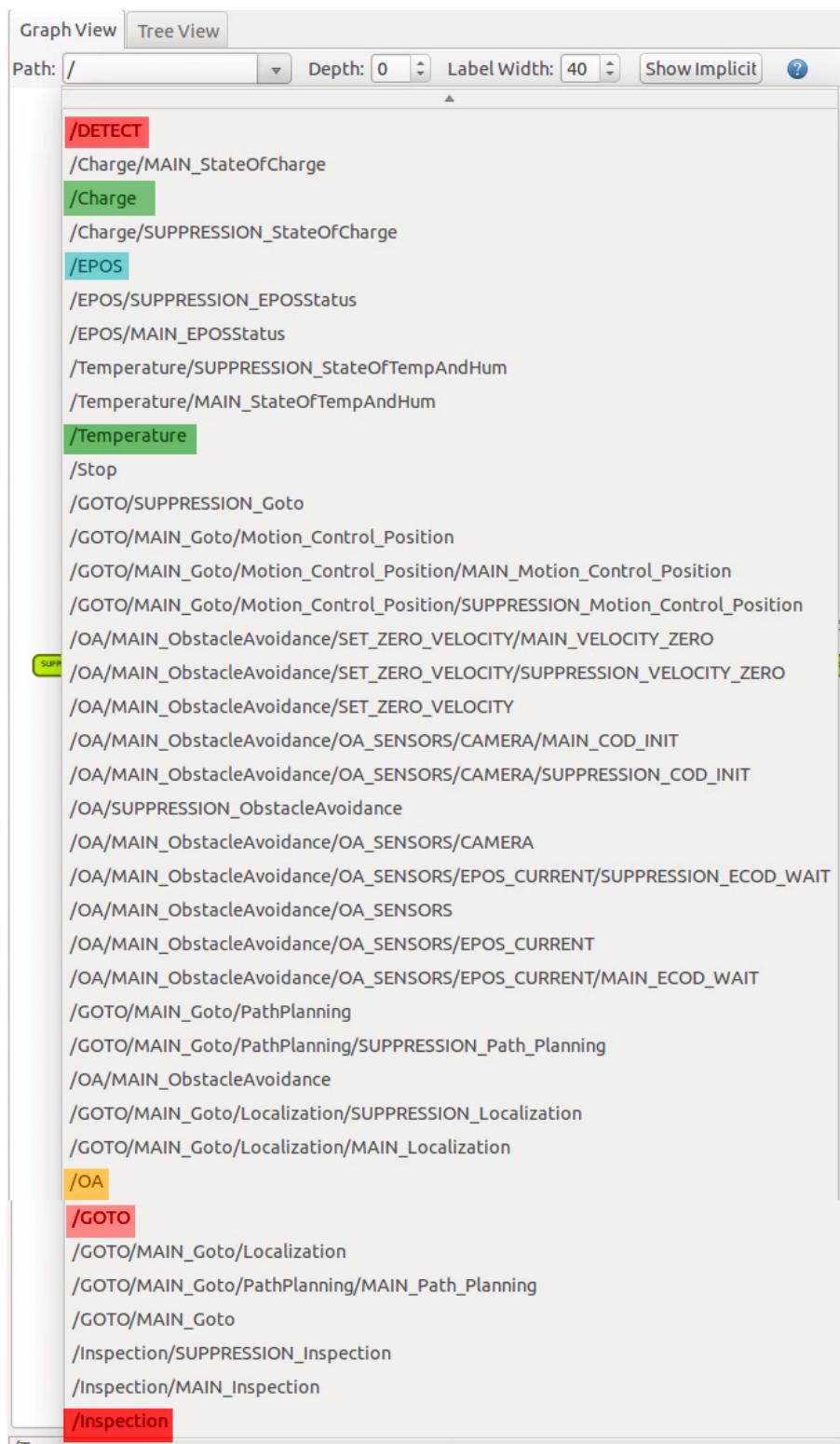


Figura 4.11: Missões e processos na execução da missão complexa INSPECTION.

Capítulo 5

Conclusões e trabalhos futuros

A dissertação “Arquitetura híbrida de um robô móvel guiado por trilhos” foi dividida em três partes principais: revisão bibliográfica, arquitetura proposta, e resultados.

A revisão bibliográfica, capítulo 2, apresentou os conceitos básicos (seção 2.1) para o entendimento da dissertação, de acordo com os principais roboticistas. É realizada uma revisão detalhada e cronológica dos paradigmas da robótica e suas principais arquiteturas: deliberativa, reativa, e híbrida, comentando vantagens, desvantagens e mostrando aplicações em robôs pioneiros, de transição e atuais. A análise das arquiteturas conclui que a arquitetura híbrida de três camadas é aquela com o maior número de aplicações complexas e bem sucedidas e que deve ser explorada para o robô de estudo, DORIS. Por fim, a revisão bibliográfica explora os diferentes ambientes de desenvolvimento de software em robótica (RDEs), apresentando vantagens e desvantagens dos principais e mais utilizados na literatura. Uma avaliação competitiva mostrou que ROS e SMACH são ferramentas excelentes para as camadas Funcional e Executivo, respectivamente, e foram exploradas na solução proposta pelo autor.

A arquitetura proposta, capítulo 3, é a aplicação do conhecimento adquirido e o que foi discutido durante a revisão bibliográfica, tentando aproveitar as vantagens da fusão de arquiteturas deliberativas e reativas, e contornando suas desvantagens. Na arquitetura híbrida de três camadas, há variadas formas de implementação das camadas e suas responsabilidades. O capítulo é dividido na implementação dessas camadas, destacando e compilando suas responsabilidades. Também são definidos novos conceitos para garantir modularidade e flexibilidade da solução, como plano

de missão, missão simples, missão complexa, tarefas, e processos reativos.

A camada Planejador é implementada com as responsabilidades controle de missão, agendador e cartógrafo. A primeira é uma interface usuário-robô que traduz os comandos e missões do nível superior ao nível Executivo, e exibe o feedback das missões ao operador. O agendador memoriza o plano de missão dentro do robô, marca um horário para execução, e possui um algoritmo de recuperação de falhas para reagendar o plano. O cartógrafo é uma compilação de mapas, gerencia os modelos do mundo, cria novos modelos para otimizar as tarefas, e os mantém atualizados com a ajuda da camada Funcional.

A camada Executivo usa SMACH como ferramenta para modelagem de tarefas do robô, e organiza a implementação nas responsabilidades sequenciador, selecionador, monitoramento e recuperação de erros, e gerenciamento de recursos, de forma a garantir modularidade, flexibilidade, facilidade de implementação e desempenho de execução. Além disso, é desenvolvida uma arquitetura de subsunção, na camada, com a ferramenta SMACH e ROS, obtendo assim as vantagens da função de coordenação competitiva para tarefas conflitantes. As desvantagens da subsunção, como *situatedness* e escalabilidade, são contornadas a partir de uma deliberação na camada.

A camada Funcional utiliza ROS como *framework* e comunicação entre componentes. Os componentes foram desenvolvidos em módulos e na forma recomendada pelos desenvolvedores de ROS e CLARAty, estado da arte na implementação Funcional. É um software orientado a objeto, obtendo assim modularidade de hardware, e estruturação apropriada de software para usar as propriedades de herança. Nesta camada, ainda é desenvolvido o algoritmo de planejamento de trajetória e velocidades (*Motion Planning*).

Ao fim do detalhamento da implementação, o documento segue para os resultados da arquitetura, onde DORIS é o estudo de caso. Novamente, dada a modularidade da solução, os testes das camadas podem ser realizados independentemente e, portanto, o capítulo 4 foi dividido pelas camadas. Uma metodologia de testes foi proposta e as camadas são avaliadas pelos critérios sugeridos por Arkin.

A avaliação de resultados da camada Funcional são testes exaustivos de componentes com os hardwares do robô. Drivers de sensores, atuadores e algoritmos

são testados, e o robô pode ser teleoperado com joystick ou por interface web. O repositório ROS e a organização de um software orientado a objeto aumentam a facilidade de implementação e passam por todos os critérios de Arkin com excelência.

A avaliação da camada Executivo é a simulação e avaliação das missões simples e tarefas modeladas, da arquitetura de subsunção implementada, e das desvantagens contornadas. Todas as responsabilidades da camada foram avaliadas e inseridas nos critérios de Arkin. Uma missão simples foi decomposta, detalhada e analisada nesta etapa, utilizando a metodologia proposta. Além disso, a ferramenta SMACH e o visualizador smach_viewer foram importantes para a solução, garantindo o paralelismo e a modularidade.

Por fim, a camada Planejador foi avaliada perante os critérios de Arkin. A metodologia imposta na implementação da camada e a organização, criação e definição de novos conceitos de missões, garantiram modularidade, paralelismo e flexibilidade da solução. Os testes mostraram o sistema autônomo integrado e em funcionamento.

Dessa forma, a dissertação é um estudo profundo no tema de arquiteturas robóticas e a tentativa de compilar ideias, criar novos conceitos e propor uma arquitetura híbrida mais geral para robôs autônomos na aplicação de inspeção, e que possuem desafios semelhantes à DORIS, isto é, AUVs e UGVs. A arquitetura se inspira na natureza, no funcionamento do corpo humano e de outros animais, para criar uma metodologia intuitiva de implementação e avaliação.

5.1 Trabalhos futuros

Os trabalhos futuros se referem às responsabilidades não finalizadas para cada camada da arquitetura híbrida proposta, à integração das camadas no robô, isto é, incluir o sistema autônomo no robô (sair do ambiente confortável da simulação), à realização de testes em outros sistemas robóticos (como novos desafios de planejamento de trajetórias), e à formalização matemática das tarefas.

Na camada Funcional, são necessárias implementações de algoritmos SLAM para o cartógrafo da camada Planejador, a integração dos diversos algoritmos de detecção de anomalias, o algoritmo de localização, e controle do manipulador.

Na camada Executivo, é necessária a modelagem de missões e tarefas em relação

ao manipulador do robô e outras funcionalidades simples. A camada Executivo já possui todas as responsabilidades finalizadas e simuladas, de forma que apenas alguns ajustes e modelagem de novas funcionalidades são necessárias.

Na camada Planejador, é necessário desenvolver a interface gráfica de usuário, e finalizar as responsabilidades de agendador e cartógrafo. O cartógrafo depende da camada Funcional e deve ser implementada concomitantemente. O agendador não é essencial para o sistema autônomo, mas importante para um robô de inspeção e outras aplicações, logo é necessária para uma arquitetura mais geral.

A implementação de tarefas com a ferramenta SMACH não possui formalismo matemático, logo não há análise de conflitos, deadlocks e avaliação das máquinas de estados. As FSMs SMACH não possuem modelagem igual à FSM padrão, logo a teoria de FSM não pode ser aplicada. Há a necessidade do desenvolvimento de um modelo matemático, de forma que o sistema não fique dependente de exaustivas simulações.

Por fim, são necessários testes da arquitetura em outros robôs, a fim de testar o critério *niche targetability* de Arkin.

Referências Bibliográficas

- [1] MURPHY, R., *Introduction to AI robotics*. MIT press, 2000.
- [2] SAKAGAMI, Y., WATANABE, R., AOYAMA, C., et al., “The intelligent ASIMO: System overview and integration”. In: *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, v. 3, pp. 2478–2483, 2002.
- [3] BELLINGHAM, J., COKELET, E. D., KIRKWOOD, W. J., “Observation of warm water transport and mixing in the Arctic basin with the ALTEX AUV”. In: *Autonomous Underwater Vehicles, 2008. AUV 2008. IEEE/OES*, pp. 1–5, 2008.
- [4] CAMPBELL, M., “Intelligent Autonomy in Robotic Systems”, *Bridge*, v. 40, n. 4, pp. 27–34, 2010.
- [5] ARKIN, R. C., *Behavior-based robotics*. MIT press, 1998.
- [6] STONE, H. S., *Introduction to computer architecture*. Sra, 1980.
- [7] MATARIC, M. J., “Behavior-based control: Main properties and implications”. In: *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, pp. 46–54, 1992.
- [8] BROOKS, R. A., “A robust layered control system for a mobile robot”, *Robotics and Automation, IEEE Journal of*, v. 2, n. 1, pp. 14–23, 1986.
- [9] SIEGWART, R., NOURBAKHSH, I. R., “Autonomous mobile robots”, *Massachusetts Institute of Technology*, 2004.

- [10] FRYXELL, D., OLIVEIRA, P., PASCOAL, A., et al., “Navigation, guidance and control of AUVs: an application to the MARIUS vehicle”, *Control Engineering Practice*, v. 4, n. 3, pp. 401–409, 1996.
- [11] BRUMITT, B. L., STENTZ, A., “Dynamic mission planning for multiple mobile robots”. In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, v. 3, pp. 2396–2401, 1996.
- [12] NORELIS, F., CHATILA, R. G., “Control of mobile robot actions”. In: *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pp. 701–707, 1989.
- [13] MORAVEC, H. P., “TOWARDS AUTOMATIC VISUAL BBSTACLE AVOIDANCE”. In: *International Conference on Artificial Intelligence (5th: 1977: Massachusetts Institute of Technology)*, 1977.
- [14] SCHEINMAN, V. D., *Design of a computer controlled manipulator.*, Tech. rep., DTIC Document, 1969.
- [15] ALBUS, J. S., “Outline for a theory of intelligence”, *Systems, Man and Cybernetics, IEEE Transactions on*, v. 21, n. 3, pp. 473–509, 1991.
- [16] ALBUS, J. S., MCCAIN, H. G., LUMIA, R., *NASA/NBS standard reference model for telerobot control system architecture (NASREM)*. National Institute of Standards and Technology Gaithersburg, MD, 1989.
- [17] WANG, F. Y., KYRIAKOPOULOS, K. J., TSOLKAS, A., et al., “A Petri-net coordination model for an intelligent mobile robot”, *Systems, Man and Cybernetics, IEEE Transactions on*, v. 21, n. 4, pp. 777–789, 1991.
- [18] SARIDIS, G. N., VALAVANIS, K. P., “Analytical design of intelligent machines”, *Automatica*, v. 24, n. 2, pp. 123–133, 1988.
- [19] MURATA, T., “Petri nets: Properties, analysis and applications”, *Proceedings of the IEEE*, v. 77, n. 4, pp. 541–580, 1989.
- [20] OLIVEIRA, P., PASCOAL, A., SILVA, V., et al., “Design, development, and testing at sea of the mission control system for the MARIUS autonomous underwater vehicle”, *Oceans MTS/IEEE*, 1996.

- [21] ARKIN, R., “Reactive Robotic Systems Ronald C. Arkin College of Computing Georgia Institute of Technology Atlanta, Georgia”, 1995.
- [22] HOLLAND, O., “Grey Walter: the pioneer of real artificial life”. In: *Proc. 5th Int. Workshop Synthesis Simulation Living Syst*, pp. 34–41, 1997.
- [23] BRAITENBERG, V., *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.
- [24] TRIBELHORN, B., DODDS, Z., “Evaluating the Roomba: A low-cost, ubiquitous platform for robotics research and education”. In: *Robotics and Automation, 2007 IEEE International Conference on*, pp. 1393–1399, 2007.
- [25] BELLINGHAM, J., GOUDEY, C., CONSI, T., et al., “A second generation survey AUV”. In: *Autonomous Underwater Vehicle Technology, 1994. AUV'94., Proceedings of the 1994 Symposium on*, pp. 148–155, 1994.
- [26] BENNETT, A., LEONARD, J. J., OTHERS, “A behavior-based approach to adaptive feature detection and following with autonomous underwater vehicles”, *Oceanic Engineering, IEEE Journal of*, v. 25, n. 2, pp. 213–226, 2000.
- [27] BOSWELL, A., LEANEY, J., “Using the subsumption architecture in an autonomous underwater robot: Expostulations, extensions and experiences”, *Proceedings of the IARP 2nd Workshop on: Mobile Robots for Subsea Environments*, pp. 95–106, 1994.
- [28] ARKIN, R. C., RISEMAN, E. M., HANSON, A. R., “AuRA: An architecture for vision-based robot navigation”. In: *proceedings of the DARPA Image Understanding Workshop*, pp. 417–431, 1987.
- [29] ARBIB, M. A., “Schema theory”, *The Encyclopedia of Artificial Intelligence*, v. 2, pp. 1427–1443, 1992.
- [30] ARKIN, R. C., MURPHY, R., PEARSON, M., et al., “Mobile robot docking operations in a manufacturing environment: Progress in visual perceptual strategies”. In: *Proc. IEEE International Workshop on Intelligent Robots and Systems*, v. 89, pp. 147–154, 1989.

- [31] BRUTZMAN, D., BURNS, M., CAMPBELL, M., et al., “NPS Phoenix AUV software integration and in-water testing”. In: *Autonomous Underwater Vehicle Technology, 1996. AUV'96., Proceedings of the 1996 Symposium on*, pp. 99–108, 1996.
- [32] KORTENKAMP, D., SIMMONS, R., “Robotic systems architectures and programming”, In: *Springer Handbook of Robotics*, pp. 187–206, Springer, 2008.
- [33] GAT, E., “Reliable goal-directed reactive control of autonomous mobile robots”, 1991.
- [34] VOLPE, R., NESNAS, I., ESTLIN, T., et al., “The CLARAty architecture for robotic autonomy”. In: *Aerospace Conference, 2001, IEEE Proceedings.*, v. 1, pp. 1–121, 2001.
- [35] HEALEY, A., MARCO, D., MCGHEE, R. B., “Autonomous underwater vehicle control coordination using a tri-level hybrid software architecture”. In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, v. 3, pp. 2149–2159, 1996.
- [36] MONTEMERLO, M., THRUN, S., DAHLKAMP, H., et al., “Winning the DARPA Grand Challenge with an AI robot”. In: *Proceedings of the national conference on artificial intelligence*, v. 21, n. 1, p. 982, 2006.
- [37] STARANOWICZ, A., MARIOTTINI, G. L., “A survey and comparison of commercial and open-source robotic simulator software”. In: *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, p. 56, 2011.
- [38] ELFES, A., TALUKDAR, S. N., *A Distributed Control System for the CMU Rover.*, Tech. rep., DTIC Document, 1983.
- [39] KRAMER, J., SCHEUTZ, M., “Development environments for autonomous mobile robots: A survey”, *Autonomous Robots*, v. 22, n. 2, pp. 101–132, 2007.

- [40] GERKEY, B., VAUGHAN, R. T., HOWARD, A., “The player/stage project: Tools for multi-robot and distributed sensor systems”. In: *Proceedings of the 11th international conference on advanced robotics*, v. 1, pp. 317–323, 2003.
- [41] QUIGLEY, M., CONLEY, K., GERKEY, B., et al., “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*, v. 3, n. 3.2, p. 5, 2009.
- [42] DASTANI, M., HÜBNER, J. F., LOGAN, B., *Programming Multi-Agent Systems: 10th International Workshop, ProMAS 2012, Valencia, Spain, June 5, 2012, Revised Selected Papers*. v. 7837. Springer, 2013.
- [43] PETERSON, J. L., “Petri net theory and the modeling of systems”, 1981.
- [44] FIRBY, R. J., “An investigation into reactive planning in complex domains.” In: *AAAI*, v. 87, pp. 202–206, 1987.
- [45] GEORGEFF, M. P., INGRAND, F. F., *Decision-making in an embedded reasoning system*. Australian Artificial Intelligence Institute, 1989.
- [46] GAT, E., “ESL: A language for supporting robust plan execution in embedded autonomous agents”. In: *Aerospace Conference, 1997. Proceedings.*, IEEE, v. 1, pp. 319–324, 1997.
- [47] SIMMONS, R., APFELBAUM, D., “A task description language for robot control”. In: *Intelligent Robots and Systems, 1998. Proceedings.*, 1998 IEEE/RSJ International Conference on, v. 3, pp. 1931–1937, 1998.
- [48] VERMA, V., ESTLIN, T., JÓNSSON, A., et al., “Plan execution interchange language (PLEXIL) for executable plans and command sequences”. In: *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2005.
- [49] ZIPARO, V. A., IOCCHI, L., “Petri net plans”. In: *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, pp. 267–290, 2006.

- [50] BOHREN, J., COUSINS, S., “The SMACH high-level executive [ROS news]”, *IEEE Robotics & Automation Magazine*, v. 4, n. 17, pp. 18–20, 2010.
- [51] ZIPARO, V. A., IOCCHI, L., LIMA, P. U., et al., “Petri net plans”, *Autonomous Agents and Multi-Agent Systems*, v. 23, n. 3, pp. 344–383, 2011.
- [52] MCCARTHY, J., HAYES, P. J., “Some philosophical problems from the standpoint of artificial intelligence”, *Readings in artificial intelligence*, pp. 431–450, 1969.
- [53] DE GIACOMO, G., LESPÉRANCE, Y., LEVESQUE, H. J., “ConGolog, a concurrent programming language based on the situation calculus”, *Artificial Intelligence*, v. 121, n. 1, pp. 109–169, 2000.
- [54] BOHREN, J., RUSU, R. B., JONES, E. G., et al., “Towards autonomous robotic butlers: Lessons learned with the pr2”. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 5568–5575, 2011.
- [55] HAREL, D., “Statecharts: A visual formalism for complex systems”, *Science of computer programming*, v. 8, n. 3, pp. 231–274, 1987.
- [56] KOLBE, F., *Goal Oriented Task Planning for Autonomous Service Robots*, Master’s Thesis, Hamburg University of Applied Sciences, 2013.
- [57] LEMBURG, J., DE GEA FERNÁNDEZ, J., EICH, M., et al., “AILA-design of an autonomous mobile dual-arm robot”. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 5147–5153, 2011.
- [58] WACHSMUTH, S., SIEPMANN, F., ZIEGLER, L., et al., “ToBI-Team of Bielefeld: The Human-Robot Interaction System for RoboCup@ Home 2012”, 2012.
- [59] CURRIE, K., TATE, A., “O-Plan: the open planning architecture”, *Artificial Intelligence*, v. 52, n. 1, pp. 49–86, 1991.
- [60] NAU, D., CAO, Y., LOTEM, A., et al., “SHOP: Simple hierarchical ordered planner”. In: *Proceedings of the 16th international joint conference on Artificial intelligence- Volume 2*, pp. 968–973, 1999.

- [61] CHIEN, S. A., KNIGHT, R., STECHERT, A., et al., “Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling.” In: *AIPS*, pp. 300–307, 2000.
- [62] MUSCETTOLA, N., NAYAK, P. P., PELL, B., et al., “Remote agent: To boldly go where no AI system has gone before”, *Artificial Intelligence*, v. 103, n. 1, pp. 5–47, 1998.
- [63] HU, G., TAY, W. P., WEN, Y., “Cloud robotics: architecture, challenges and applications”, *Network, IEEE*, v. 26, n. 3, pp. 21–28, 2012.
- [64] CARVALHO, G., FREITAS, G., COSTA, R., et al., “Doris-monitoring robot for offshore facilities”. In: *Offshore Technology Conference Brasil*, 2013.
- [65] FREITAS, R. S., XAUD, M. F., MARCOVISTZ, I., et al., “THE EMBEDDED ELECTRONICS AND SOFTWARE OF DORIS OFFSHORE ROBOT”, *IFAC-PapersOnLine*, v. 48, n. 6, pp. 208–213, 2015.
- [66] HUNZIKER, D., GAJAMOHAN, M., WAIBEL, M., et al., “Rapyuta: The roboearth cloud engine”. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 438–444, 2013.
- [67] CORRIGAN, S., “Introduction to the controller area network (CAN)”, *Texas Instrument, Application Report*, 2008.
- [68] CHRISTENSEN, L., FISCHER, N., KROFFKE, S., et al., “Cost-effective autonomous robots for ballast water tank inspection”, *Journal of ship production and design*, v. 27, n. 3, pp. 127–136, 2011.
- [69] MARTIN, S. C., WHITCOMB, L. L., YOERGER, D., et al., “A mission controller for high level control of autonomous and semi-autonomous underwater vehicles”. In: *OCEANS 2006*, pp. 1–6, 2006.