TOLERÂNCIA A FALHAS ADAPTATIVA PARA ROBÔS MÓVEIS COM ARQUITETURA HíBRIDA

Por

Wilton Speziali Caldas

DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO
NA
UNIVERSIDADE FEDERAL DE MINAS GERAIS
BELO HORIZONTE, MINAS GERAIS
MARÇO 2004

Resumo

Este trabalho apresenta uma metodologia de desenvolvimento do controle para robôs móveis de arquitetura híbrida integrando tolerância a falhas adaptativa. A realização de tarefas críticas ou perigosas por robôs torna a tolerância a falhas um requisito fundamental. Os robôs devem ser capazes de realizar as tarefas desejadas mesmo na presença de defeitos, implementando a tolerância a falhas através do uso das redundâncias que existam em um único robô ou distribuída nas diversas competências individuais dos times cooperativos. Na concepção de robôs, principalmente no caso dos autônomos, é necessário se trabalhar com muitas restrições: o hardware não é limitado apenas pelo custo, mas também pela possibilidade de ser incorporado ao robô móvel; o consumo dos componentes limita a autonomia do sistema; a percepção e controle devem ser realizados em tempo real. Com estas várias restrições, a tolerância a falhas ideal deve maximizar a disponibilidade e confiabilidade total do sistema nestes sistemas e permitir o uso inteligente dos recursos disponíveis em qualquer situação. O controle do robô deve se adaptar as condições ambientais, condições do hardware e do software e aos objetivos da tarefa ou missão específica que esta sendo realizada. A metodologia desenvolvida, a qual por ser automatizada, facilita o processo de síntese das políticas de adaptação e a integração destas ao controle de forma maximizar a confiabilidade ou o desempenho em função do estado global do sistema.

Abstract

This work presents a novel architectural design methodology, which enables weaving fault-tolerance into hybrid architecture control frameworks. The beneficial aspects fostered by fault tolerance greatly surpass the overhead in project development. Dataflow processing paradigm is based on functions and abstract data elements, providing a simple, yet powerful, description power. The simple structure allows the inclusion of redundant information and processing elements within the control. Redundancies elicited by dataflow automatically enable alternative configurations of available functional blocks. The methodology also gives the designer the freedom to define requirements and restrictions for both control performance and reliability. A control graph contains all the information necessary to optimize the robot resources usage and also captures all the necessary adaptations for the system. This graph may be automatically generated combining the hybrid control description and the dataflow information. The thesis shows the results from a prototype of hybrid architecture for a Nomad robot. The resulting fault tolerant hybrid architecture provides real time execution of control and fault recovery. Experiments were carried out to demonstrate the gain in reliability versus the added overhead.

Agradecimentos

Esta tese foi uma realização pessoal e um desafio, o qual gostei de enfrentar. O resultado que obtive ao final deste trabalho é muito maior que a contribuição científica para a comunidade, mas um crescimento e amadurecimento tanto pessoal quanto profissional. A experiência de se desenvolver uma tese pode-se dizer que no mínimo é estressante. A necessidade de se realizar um trabalho significativo para a comunidade, o qual será avaliado por pessoas do mais alto nível, preenche grande parte do doutorado com inseguranças, duvidas e sempre muita auto-avaliação. A complexidade do trabalho exige além dos conhecimentos técnicos, o uso de métodos de trabalho e a persistência para se avançar na direção de uma meta de grande importância, com pequenos passos e pequenas realizações a cada dia. O comprometimento necessário para o trabalho é tão grande que envolve as pessoas de sua convivência, exigindo por muitas vezes sacrifícios por parte delas. Gerenciar as relações pessoais com pouco tempo disponível torna sempre clara a preciosidade existente em cada momento. Claramente, a realização desta meta não seria possível sem a grande ajuda e paciência de todos a minha volta. Agradeço em especial a minha esposa Ana Maria, uma mulher muito especial, que além de sempre me apoiar, dedicou-me muita compreensão e paciência e de tempo para estar ao meu lado. Agradeço a minha família que sempre me apoiou em minhas decisões e compreendeu muitas vezes minhas omissões na atenção a eles devida. Agradeço em especial a minha mãe, Maria Rosa e ao meu pai, Joaquim que sempre tiveram a educação dos filhos como uma meta primordial e sacrificaram-se muito por este ideal. Aos meus irmãos Wilson e Wagner, que além de me apoiarem, ajudaram-me a compreender e a enfrentar os obstáculos de cada

dia. Aos meus amigos e amigas, que além de ajudarem-me a divertir, perdoaram-me muitas vezes pela minha falta de tempo para as conversas. Agradeço aos meus orientadores Antônio Otávio Fernandes e Mário Fernando Montenegro Campos, que além de me conduzirem pelo mundo acadêmico foram grandes amigos. Agradeço em especial a Deus, por ter recebido tanto e por Sua presença em cada momento da minha vida. Agradeço a Ele por ter encontrado minha esposa, por toda minha família, meus amigos, e pela oportunidade de ter desenvolvido este trabalho. Enfim por eu ser muito feliz.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A verdadeira origem da descoberta consiste não em procurar novas paisagens, mas em ter novos olhos.

Marcel Proust(1871-1922)

1.1 Motivação

No cenário atual existe uma crescente demanda por sistemas robóticos que possuam alta disponibilidade e confiabilidade. Entre as novas aplicações destacamos algumas: cirurgias de alta precisão, em que um erro pode comprometer a saúde ou a vida de um paciente de forma permanente; tarefas domésticas e de apoio a idosos e a enfermos na própria residência, onde a interação do homem com a máquina é uma constante, e qualquer erro, como uma colisão, pode significar um grande risco pessoal [Koji et al., 2001]. De maneira geral, a confiabilidade e muitas vezes a disponibilidade são características essenciais ao executar tarefas que envolvam interação com os seres humanos. Em outro extremo estão as tarefas em ambientes hostis ou de difícil acesso, nas quais a presença humana é impossível ou muito perigosa, como: limpeza de lixo radioativo; trabalho em uma plataforma de petróleo no fundo do mar; tarefas no espaço, como reparo de satélites ou exploração de um planeta remoto.

O emprego da robótica nessas áreas requer dos sistemas robóticos características essenciais como a segurança, a confiabilidade e a disponibilidade [Visinsky, 1994, Cavallaro and Walker, 1994, Koji et al., 2001, Huntsberger et al., 2000]. A confiabilidade e a disponibilidade de um sistema não dependem apenas da probabilidade de falhas, mas também do custo e do tempo para se realizar um reparo, e muitas vezes

da possibilidade de acesso e de intervenção no sistema defeituoso. Estes fatores tornam cada vez mais importantes a tolerância a falhas nos robôs, a fim de reduzir a necessidade de intervenção humana.

A implementação de tolerância a falhas em qualquer tipo de sistema é um processo complexo [Cavallaro and Walker, 1994]. Alguns aspectos da robótica móvel a tornam ainda mais difícil, como por exemplo: as restrições de tempo real, decorrentes da interação com o mundo; as incertezas inerentes aos sensores e atuadores; e a interação do robô com o meio ambiente, que é de difícil modelagem e, muitas vezes, imprevisível. Garantir a confiabilidade de um robô é um desafio, pois envolve tolerar falhas sensoriais, falhas mecânicas e possíveis falhas no controle.

A implementação da tolerância a falhas sempre depende da existência de algum tipo de redundância [Somani and Vaidya, 1997]. A tolerância a falhas sensoriais e a falhas mecânicas é possível apenas com a existência de redundância específica e individualizada durante o projeto [Ferrell, 1993]. Já a redundância no controle pode ser obtida utilizando métodos mais abrangentes, adequados a falhas de software ou a falhas no processamento em sistemas de tempo real [Kalbarczyk et al., 1999]. O sucesso da aplicação destes métodos genéricos no controle de robôs móveis deve considerar alguns fatores:

- Os recursos disponíveis são normalmente limitados devido a restrições construtivas, restrições de consumo ou autonomia, ou restrições de peso ou custo. Estas restrições são ditadas pelos requisitos de mobilidade, que limitam a disponibilidade de recursos redundantes.

- Os mecanismos de tolerância para sensores e atuadores podem alterar significativamente a demanda de processamento do sistema, a qual pode ser bastante diferente na presença de falhas.

- Os requisitos de desempenho e confiabilidade nos processos de controle ou em outros elementos do sistema podem variar significativamente ao longo do tempo, em função dos objetivos correntes da missão ou de sua fase. Por exemplo, um robô móvel que percorre um ambiente recolhendo lixo tóxico pode concentrar o uso de seus recursos de processamento na função de navegação quando está procurando os itens a serem recolhidos. Após encontrar um item, pode concentrar o uso de seus recursos para manipulação do lixo.

Da mesma forma que nos robôs móveis muitos dos sistemas que requerem alta disponibilidade e confiabilidade, possuem também, outras fortes restrições construtivas ou de custo. Os satélites, naves e sistemas autônomos em geral possuem restrições de peso, de consumo ou de processamento. Portanto, a utilização dos recursos disponíveis, que normalmente são escassos, deve ser otimizada.

A tolerância a falhas adaptativa é a adequação dinâmica da confiabilidade e do desempenho do sistema completo (ou de módulos específicos), em função de variações das condições externas, internas ou dos objetivos correntes da missão. Ela permite que o sistema, num dado momento, concentre a utilização dos recursos disponíveis nos elementos mais críticos para o sucesso da tarefa corrente. A tolerância a falhas adaptativa para falhas de software ou de processamento surgiu na última década [Kim and Lawrence, 1992], sendo normalmente utilizada em sistemas multiprocessados de tempo real, que necessitam de grande capacidade de processamento, além da tolerância a falhas [Hecht et al., 2000, Shokri et al., 1998].

Um controle que implemente a tolerância a falhas adaptativa, seleciona uma configuração do sistema levando em conta requisitos de confiabilidade, desempenho, estados internos, falhas detectadas, e a percepção do ambiente. Cada adaptação define o uso de recursos de hardware e software, podendo ser: sensores e atuadores, comunicação, versões de software, processadores, baterias ou qualquer outro elemento configurável do robô.

Os conceitos existentes na tolerância a falhas adaptativa são valiosos para a utilização em controles como dos robôs móveis, independentemente das tecnologias empregadas em um dado momento. Entretanto os métodos e sistemas encontrados na literatura para tolerância a falhas adaptativa focam apenas os problemas associados as falhas de processamento. O tratamento dado aos outros tipos de falhas e reestruturação do processamento das informações continua sendo implementado de forma personalizada e com alto nível de especificidade. A falta de metodologias e ferramentas mais genéricas que facilitem o projeto de sistemas móveis com tolerância a falhas adaptativa motivou o desenvolvimento deste trabalho.

1.2 Contribuição da Tese

A principal contribuição desta tese é o desenvolvimento de uma metodologia que facilita o processo de inserção de tolerância a falhas adaptativa no controle de um robô móvel. A metodologia foi desenvolvida com o intuito de fornecer mecanismos padronizados de detecção e recuperação de falhas, disponíveis em bibliotecas, além de permitir ao projetista a inclusão de soluções específicas. Além disso, o processo de desenvolvimento do controle proposto permite que determinadas etapas sejam automatizadas, possibilitando a síntese das políticas de redundância necessárias à implementação da tolerância a falhas adaptativa.

A metodologia desenvolvida, para a tolerância a falhas adaptativa, oferece uma abordagem mais genérica do que outros trabalhos existentes na literatura no tratamento de informações redundantes, detecção e isolamento de falhas. Além disso, propõe uma estrutura de controle híbrida que integra no alto nível uma máquina de estados finitos com um controle adaptativo.

O trabalho visa prover aos projetistas recursos de tolerância a falhas, sem restringir em demasiado a sua liberdade na programação das funções de controle. Muitas abordagens de controle de robôs foram analisadas cuidadosamente para que o modelo desenvolvido fosse o mais genérico possível. A arquitetura híbrida com o controle de alto nível implementado por uma máquina de estados finitos foi selecionada para o modelo, devido a sua simplicidade, além de facilitar alterações e extensões automáticas. Os estados da máquina ativam funções de controle de mais baixo nível, para as quais existem várias abordagens diferentes. Para os nossos propósitos, consideramos como aplicáveis ao mais baixo nível do modelo as seguintes abordagens: comportamentos [Brooks, 1999] e [Mataric, 1997]; esquemas perceptivos e esquemas motores [Arkin, 1998] e [Murphy and Hershberger, 1996]; sistemas híbridos (conjunto de equações contínuas associadas a estados discretos) [Chaimowicz et al., 2001].

Do ponto de vista do projetista, a metodologia facilita a criação e o uso de múltiplas configurações (adaptações) de forma simples e eficiente, permitindo que ele se concentre na programação das funções de controle e nos requisitos necessários de desempenho e confiabilidade. Uma estrutura padronizada de implementação permite a síntese automática das adaptações necessárias. Junto com o modelo de implementação foi desenvolvida uma plataforma de controle adaptativo, que incorpora a máquina de estados do controle híbrido a ativação do controle de baixo nível e os recursos básicos de detecção e recuperação de falhas. Além da plataforma, é oferecida uma biblioteca, que pode ser ampliada pelo projetista, contendo funções de acesso à plataforma de controle, primitivas de comunicação entre os blocos e algumas funções primitivas para detecção de falhas.

O controle do robô é implementado por uma arquitetura híbrida na qual o projetista define missões, as quais são divididas em fases. A cada fase é associado um escalonamento de blocos de código, chamados de Blocos Funcionais (*BFs*), que implementam o controle de mais baixo nível. Os *BFs* se interconectam através de canais abstratos chamados de Elementos de Dados (*EDs*). O projetista descreve a conexão dos *BFs* e *EDs* de forma a criar um fluxo de dados. Caso exista redundância neste fluxo, é possível criar subconjuntos distintos correspondentes às possíveis adaptações do sistema. Cada uma das adaptações é analisada sob aspectos de confiabilidade, de tolerância a falhas e de desempenho na execução da missão. As adaptações que oferecerem um ganho significativo são selecionadas criando as políticas necessárias para a Tolerância a Falhas Adaptativa [Kim, 2000]. Os critérios da adaptação são definidos utilizando os requisitos de desempenho e de confiabilidade associados a cada fase de uma missão. O fluxo de dados responsável pelo processamento de baixo nível representa diretamente uma estrutura de interdependência de informações, o que facilita o cálculo de confiabilidade de cada uma das adaptações, simplificando a implementação da tolerância a falhas adaptativa.

Este trabalho apresenta uma metodologia para desenvolvimento de sistemas

robóticos com tolerância a falhas adaptativa que restringe a forma de implementação de nível mais baixo a um fluxo de dados, entretanto, sem restringir a abordagem utilizada para o controle. Esta é uma das características que o torna mais genérico que outros trabalhos realizados [Bagchi et al., 1998, Hecht et al., 2000, Kim and Lawrence, 1992]. Além disso, as estruturas regulares utilizadas facilitam a automatização do processo de síntese das políticas de redundância necessárias a tolerância a falhas adaptativa, permitindo assim o desenvolvimento de um *framework* específico.

A Plataforma de Controle Adaptativo (*PCA*), as bibliotecas e o protótipo foram desenvolvidos em linguagem *C* no sistema operacional Linux. É importante ressaltar que a metodologia desenvolvida é aplicável a outras linguagens, inclusive com orientação a objetos.

1.3 Estrutura do texto

O Capítulo 2 introduz o assunto de tolerância a falhas. O Capítulo 3 descreve algumas arquiteturas de controle utilizadas para robôs móveis. O Capítulo 4 descreve trabalhos em tolerância a falhas de robôs que foram utilizados como base para o modelo desenvolvido. Uma visão geral do modelo é apresentada no Capítulo 5. No Capítulo 6 é descrito o fluxo de processamento de baixo nível, e no Capítulo 7 é descrito o controle de alto nível. A formação do grafo de controle e alguns detalhes de implementação são apresentados no Capítulo 8. No Capítulo 9 são apresentados detalhes do protótipo e os resultados obtidos. Para finalizar o trabalho, as conclusões e trabalhos futuros são apresentados no Capítulo 10.

Capítulo 2

Tolerância a falhas

> Se alguma coisa pode dar errada, dará.
>
> Lei de Murphy

A tolerância a falhas é um campo de pesquisa bastante amplo e maduro, entretanto a sua implementação está sempre intrinsecamente associada a uma aplicação ou a um sistema específico. Não existe uma solução única e mágica capaz de resolver todos os variados problemas de confiabilidade e disponibilidade, porque sempre há uma característica ou peculiaridade própria e individual de cada sistema e aplicação.

A tolerância a falhas pode ser definida informalmente como a capacidade de um sistema de concluir uma tarefa determinada na presença de defeitos de hardware ou de software. Neste texto serão usadas as seguintes definições básicas [Barreto and Fernandes, 1997]:

Defeitos: são problemas físicos reais que ocorrem no hardware como um curto circuito ou um motor travado. Existem também defeitos de software, como codificações incorretas que provocam erros na execução de programas.

Falhas: são as manifestações de um defeito no sistema. Pode existir uma linha incorreta em um software e esta nunca ser executada. Neste caso, existe o defeito, sem que haja a sua manifestação na forma de uma falha. As falhas podem ocorrer tanto em software quanto com o hardware.

Erros: são as manifestações das falhas observadas no sistema. Acontecem quando o comportamento percebido do sistema não é o esperado, ou seja, suas saídas não são corretas.

A *tolerância a falhas* pode ser definida como a capacidade de um sistema fornecer as suas saídas corretas, isto é, saídas sem erros, mesmo na presença de defeitos e falhas.

A implementação de métodos de tolerância a falhas é, normalmente, um processo extremamente complexo para qualquer tipo de sistema. É muito difícil prever todas as condições e estados externos e internos, a que o sistema será submetido durante sua utilização.

A invariante básica na implementação de tolerância a falhas, em qualquer tipo de sistema, é a existência de algum tipo de redundância, seja esta no controle, na informação processada, no método para realizar a tarefa desejada, ou na existência de uma cópia do sistema inteiro ou de partes deste [Somani and Vaidya, 1997]. Infelizmente, a redundância pode ser limitada por fatores de custo ou por fatores tecnológicos.

Quando se fala de tolerância a falhas é importante definir também de forma precisa qual é o funcionamento correto esperado para o sistema e conseqüentemente, quais são os possíveis erros indesejáveis. O *modelo de falhas* é outra especificação fundamental, sendo a previsão do conjunto de defeitos e as falhas, que o sistema é capaz de suportar sem que este apresente erros. Para se tolerar as falhas, estas devem ser isoladas ou contidas de forma a não prejudicar o funcionamento global esperado do sistema.

A tolerância a falhas está diretamente ligada à aplicação do sistema, ao seu projeto, à definição do ambiente esperado para o funcionamento e aos requisitos de confiança e disponibilidade esperados.

Os dois índices mais comuns de se expressar a habilidade do sistema em tolerar falhas é a confiabilidade e disponibilidade (*Reliability and Availability*) [Somani and Vaidya, 1997]. A confiabilidade é a probabilidade do sistema permanecer funcionando corretamente durante toda a duração da missão. Uma confiabilidade muito alta é desejada em situações onde a manutenção é indesejável ou impossível. É um requisito essencial em aplicações críticas como viagens espaciais ou controle industrial, onde uma falha pode significar perda de vidas. Também é essencial quando o custo de manutenção pode ser muito elevado. O reparo de um satélite ou a perda de uma missão a Marte pode representar um prejuízo muito grande.

A disponibilidade expressa a fração do tempo em que um sistema está operacional. Uma disponibilidade de 0.999999 para uma missão de 10 horas significa que a probabilidade de falhas durante a missão pode ser, no máximo, 10^{-6}. É importante notar que sistemas com alta disponibilidade podem falhar, desde que a freqüência da ocorrência das falhas, e o tempo de recuperação sejam pequenos o suficiente para garantir a disponibilidade desejada. Este é o caso de servidores de comércio eletrônico, venda de passagens e outros, nos quais a não disponibilidade do serviço pode significar grandes perdas financeiras. Existem ainda outros valores, correlacionados a confiabilidade e disponibilidade, utilizados para análise.

- *Probabilidade de Falhas*: É a probabilidade de um determinado componente ou módulo apresentar falhas em um dado instante de tempo ou período. Dada uma

constante de falhas γ e uma distribuição exponencial a probabilidade de falhas é $p(t) = 1 - e^{-\gamma t}$.

- *Tempo médio entre falhas* ou o MTTF (*Medium Time to Failure*) é representado por $MTTF = \frac{1}{\gamma}$.
- *Confiabilidade* $R(t) = 1 - p(t) = e^{-\gamma t}$.
- *Tempo médio de reparo.*
- *Disponibilidade* do sistema que é uma função do MTTF e do tempo de reparo.

Estabelecer a confiabilidade e disponibilidade de um sistema não é um problema simples. Uma possibilidade é coletar informações de falhas de um número significativo de sistemas, nas condições de funcionamento esperado, e realizar análises estatísticas a partir destes dados. Entretanto, se a quantidade necessária de elementos para se coletar informações relevantes for muito grande, ou o tempo esperado da missão for muito longo, este processo pode ser inviável. A outra solução possível é criar modelos matemáticos ou simulações para inferir os dados de confiabilidade e disponibilidade de um sistema em função das informações de cada elemento que o compõem. Os parâmetros obtidos desta forma podem ser refinados através da observação constante dos sistemas.

O planejamento para se evitar a ocorrência de falhas (*fault avoidance*) é um aspecto muito importante de um projeto tolerante a falhas [Somani and Vaidya, 1997]. Ele se inicia com a especificação de requisitos e a análise do ambiente e das falhas que devem ser toleradas para se obter a confiabilidade necessária.

Definir quais as falhas que o sistema vai suportar, é definir o modelo de falhas. Este determina quais são os conjuntos possíveis de defeitos e falhas para os quais o sistema vai continuar funcionando sem apresentar erros. Muitas vezes, é aceitável a degradação de desempenho ou de serviços, outras vezes não. Por isso, o modelo de falhas é intimamente associado aos requisitos de funcionamento do sistema, ou seja, aplicação do sistema.

2.1 O uso de redundância

Manter o funcionamento de um sistema na presença de uma falha significa possuir mais de uma opção para se realizar o serviço desejado. A existência de redundância no sistema é o princípio básico do projeto tolerante a falhas. A redundância de qualquer sistema sempre tem um custo direto, seja este financeiro ou de tempo. O projeto de um sistema envolve um compromisso entre a redundância utilizada, o custo e o nível de tolerância obtida, visando sempre otimizar esta relação entre o custo e o benefício.

Existem três tipos de redundâncias básicas: a redundância espacial, a de informação e a temporal.

A redundância espacial corresponde à existência de um hardware repetido para realizar uma determinada função. A redundância de informação pode ser de muitas formas diferentes como o uso de dados provenientes de fontes diferentes, o uso de backup dos dados, ou até o uso de códigos de detecção de erros. A redundância temporal corresponde à repetição de um processamento ou tarefa ao longo do tempo

A redundância temporal implica, normalmente, em tempo maior para detecção e recuperação de uma falha, quando comparada à redundância espacial. Por outro lado, a redundância espacial normalmente aumenta o custo de hardware, peso e consumo de recursos. Em um sistema tolerante a falhas é usual existir dois ou mais tipos de redundância trabalhando em conjunto.

A existência de redundância é condição necessária, mas não é suficiente para existir a tolerância à falhas. O sistema deve manter o comportamento ou saídas corretas mesmo na presença de falhas. É essencial, portanto, que o efeito das falhas seja isolado ou contido de maneira que o funcionamento do sistema não seja perturbado. O método utilizado para isolar ou recuperar a falha pode variar e é parte fundamental do projeto do sistema. Para se efetuar o isolamento e a recuperação de falhas é normalmente necessária a sua detecção, e, algumas vezes, a realização de um diagnóstico preciso.

A essência do processo de detecção de falhas é determinar quando o resultado gerado por um módulo está incorreto, seja este módulo composto por software, hardware ou ambos. Detectada a falha, o módulo que a gerou é ignorado, reinicializado, calibrado ou sofre qualquer outra ação corretiva. Existem alguns métodos clássicos para se implementar em sistemas a tolerância a falhas de computação:

Redundância Modular: São utilizadas múltiplas réplicas idênticas do hardware e um mecanismo de votação. Cada módulo replicado executa as mesmas funções e envia os resultados para o mecanismo de votação. Este determina a provável saída correta ao selecionar o resultado mais votado. É importante ressaltar, que o mecanismo de votação deve apresentar a confiabilidade extremamente alta, possuindo internamente recursos de tolerância a falhas.

Programação utilizando N-versões: Este método pode tolerar falhas em hardware e software. Várias versões do mesmo módulo de software são implementadas por equipes diferentes. Um mecanismo de votação recebe as saídas dos diversos módulos e seleciona o resultado correto.

Codificação com controle de erros: Uma replicação completa é efetiva, mas é normalmente muito cara. Para certas aplicações como memória ou barramento de dados é necessária uma redundância menor que a replicação completa. Neste caso, utilizam-se códigos que permitem a detecção e, algumas vezes, recuperação

de falhas. Os códigos paridade, Hamming e códigos não ordenados são usados freqüentemente.

Pontos de Controle e Retorno (*Checkpoints and Rollbacks*): Uma cópia estável do estado de um sistema ou módulo é salva em algum armazenamento imune às falhas consideradas, constituindo este, um ponto de controle. Um retorno é executado a partir do último ponto de controle salvo quando uma falha é detectada. Esta técnica é aplicável tanto a falhas de hardware, quanto a certos tipos de falhas de software.

Blocos de Recuperação: Utilizam-se múltiplas alternativas para realizar a mesma função. Um bloco é primário e os outros são secundários. Quando o bloco primário termina a sua execução, um processo de validação do resultado é efetuado. Caso seja detectada uma falha, os blocos secundários são executados um a um até que não existam mais alternativas, ou até que o resultado correto seja obtido. Neste caso, alguma invariante é utilizada para se avaliar os resultados dos blocos primários ou secundários.

Avaliação de fidelidade (*Dependability evaluation*): Um sistema tolerante a falhas projetado deve ser avaliado em função dos seus requisitos iniciais e da confiabilidade obtida. Para isto, são utilizados os modelo analíticos e os métodos de injeção de falhas. Os modelos analíticos, como por exemplo, as cadeias de Markov, permitem aos projetistas a análise dos estados possíveis do sistema e as probabilidades de cada transição. Estes modelos permitem a análise das dependências de um sistema utilizando várias métricas diferentes. Infelizmente, os métodos analíticos podem ser muito complexos e inviáveis se o modelo de descrição for muito refinado. A injeção de falhas pode ser executada em sistemas reais ou simulações. Embora seja um método genérico, deve se ter muito cuidado na avaliação dos resultados e na preparação dos padrões de teste, para se obter resultados significativos.

É importante destacar que o mecanismo de detecção de falhas, ou arbitragem dos resultados corretos, também é sujeito a defeitos. Este é projetado normalmente da forma mais robusta possível, atendendo os requisitos do sistema completo.

A recuperação das falhas é um mecanismo normalmente integrado ao processo de detecção e diagnóstico das falhas. A detecção de falhas utilizada em sistemas digitais e programa são realizados basicamente de três formas:

1. Módulos diferentes realizam a mesma tarefa e espera-se, que na ausência de falhas o resultado seja sempre o mesmo. Neste caso, o número de módulos é escolhido de acordo com a redundância desejada. É o método mais genérico,

mas exige no mínimo a utilização de três módulos equivalentes. Utilizado na redundância modular e n-versões.

2. As saídas de um módulo são divididas em um conjunto de valores válidos e um conjunto de valores inválidos, normalmente utilizando códigos com determinadas propriedades. No funcionamento correto as saídas vão sempre fazer parte do conjunto válido. Na presença de uma falha tolerada pelo sistema a saída será a correta ou será uma saída que pertencente ao conjunto de saídas invalidas. A probabilidade do sistema fornecer uma saída errada que pertencente ao conjunto válido é baixa, o que possibilita a detecção de falhas no módulo. A redundância existe internamente ao módulo, sendo utilizada nos métodos de codificação com controle de erros.

3. Existe uma invariante possível de ser verificada nas saídas corretas de um módulo. Quando existe uma falha pertencente ao modelo a invariante da saída é violada, permitindo assim a sua detecção. A invariante pode depender do estado anterior do sistema, sendo esta opção mais genérica que a apresentada no item anterior. Os testes de invariantes são utilizados de maneira geral nos métodos de Pontos de Controle e nos Blocos de recuperação.

2.2 Detecção de falhas

A detecção de falhas é realizada essencialmente através da comparação dos valores obtidos de um módulo com um conjunto de valores esperados. Quando se encontram valores discrepantes, se detectam falhas. Esta comparação pode ser dividida em duas classes: as comparações exatas e as aproximadas. As comparações exatas são adequadas aos sistemas digitais e a maioria dos métodos de tolerância a falhas em software. As comparações aproximadas são utilizadas normalmente quando o sistema envolve algum módulo analógico.

Os sistemas que não são totalmente digitais normalmente interagem com elementos do mundo real. Esta interação é realizada através de interfaces analógicas e muitas vezes mecânicas, as quais estão sempre sujeitas a erros ou incertezas. Portanto, as comparações realizadas para detecção de falhas nesta classe de sistemas devem considerar sempre a presença de erros, e conseqüentemente, comparar os valores obtidos com os valores esperados utilizando faixas ou limites de tolerância.

A definição dos valores dos limites para as comparações é um problema também muito importante. Em muitos casos é difícil diferenciar entre um erro inerente ao módulo e uma falha que possa estar acontecendo. Os limites de comparação devem ser estreitos o suficiente para detectar as falhas previstas no momento em que ocorrem, e devem ser suficientemente abertos para conter a incerteza inerente ao módulo evitando

a detecção de falsas falhas. O cálculo destes limites não é uma tarefa simples, porque envolve o conhecimento do comportamento do erro inerente ao módulo, o qual pode estar associado a elementos que funcionam com um grau de incerteza muito grande. Sensores e atuadores eletromecânicos se enquadram na classe de sistemas que operam com incertezas.

As falhas podem ser permanentes ou transientes. As permanentes apenas são corrigidas após um processo de reparo no sistema. As falhas transientes podem ocorrer por influências internas ou externas ao sistema. Quando a influência termina, o módulo pode retornar a operação normal. Algumas vezes pode ser necessário um processo para reiniciar o funcionamento normal de um módulo afetado por uma falha transiente.

Módulos que interagem com o mundo real como os sensores estão sujeitos a determinados tipos de falhas [Ferrell, 1994, Murphy and Hershberger, 1999]:

1. Erros instantâneos ou transientes - Correspondem a valores discrepantes no meio de valores coerentes. Se estas falhas duram um tempo muito pequeno em relação à necessidade do sistema, são normalmente desprezadas sem maiores impactos. Entretanto, o projeto deve ser muito cuidadoso para não executar ações inadequadas em função dos valores discrepantes.

2. Falhas de Calibração - Muitos elementos como os sensores, podem sofrer variações no comportamento, e consequentemente, variações nas suas saídas em função do tempo ou de condições de operação. Se o erro calculado entre os valores esperados e os obtidos do módulo possuir uma variação constante ao longo do tempo, pode ser possível corrigir a falha através do uso de um fator de correção. Este processo de recalibração pode ser aplicado a alguns tipos de sensores.

2.3 Diagnóstico

Além da detecção de falhas, muitas vezes é necessário o processo de diagnóstico, que é a identificação do defeito que causou a falha. A precisão do diagnóstico varia de acordo com os requisitos do sistema e do objetivo de sua utilização. Os usos mais comuns do diagnóstico são: auxílio ao projeto; auxílio ao reparo e a própria tolerância ou recuperação de falhas.

Para auxiliar um projeto, o diagnóstico pode ser utilizado durante o desenvolvimento de um sistema como uma ferramenta de depuração permitindo a identificação de pontos falhos nos protótipos. Além disso, o uso do diagnóstico nos itens produzidos é um recurso muito importante para a melhoria da qualidade, pois pode evidenciar os elementos mais sensíveis.

O reparo ou a recuperação de um sistema com falhas é realizado normalmente com o ajuste ou a substituição do elemento defeituoso. Assim sendo, a identificação correta do defeito no sistema é um passo necessário que influi diretamente no tempo e no custo da intervenção realizada. O nível necessário de refinamento do diagnóstico é muito variado e dependendo da granularidade dos elementos substituíveis e do custo da interrupção do sistema.

O diagnóstico é importante para a tolerância a falhas por dois motivos principais: O primeiro motivo é que a seleção da ação de recuperação adequada a uma determinada falha detectada pode ser dependente do diagnóstico exato do defeito. Segundo, se a falha pode ser originada de vários módulos diferentes e o defeito não for assinalado a um elemento específico, todos os módulos sob suspeita devem ser isolados. Esta atitude pode representar uma grande perda dos recursos disponíveis do sistema, proporcionando uma perda de desempenho e muitas vezes uma perda dos serviços ou tarefas oferecidos.

No diagnóstico que visa o reparo, o fator mais importante é a identificação do elemento que será substituído, e as restrições de tempo são basicamente relativas ao processo de intervenção no sistema. No diagnóstico realizado como parte do processo de tolerância a falhas, as restrições de tempo são normalmente mais rígidas. Se o diagnóstico for essencial para o processo de recuperação da falha, este passa a ser um fator crítico no tempo de resposta do sistema, muitas vezes restrito a parâmetros de execução em tempo real. Pode ser necessário levar o sistema a um estado seguro, no qual o diagnóstico é efetuado, antes que possa se dar continuidade à tarefa que estava sendo realizada, caso seja possível.

O diagnóstico envolve três diferentes espaços: o espaço de falhas que contém todas as possíveis falhas que podem ocorrer com o sistema; o espaço de observação que contém todas as observações ou informações de estado do sistema e do ambiente; o espaço de diagnóstico que contém todos os possíveis diagnósticos em função do espaço de observação.

O espaço de falhas depende da complexidade do sistema e da sua interação com o ambiente. Por exemplo, o espaço de falhas de robôs autônomos, devido à natureza imprevisível do ambiente, tende ao infinito. É praticamente impossível predizer e se precaver para todas as possíveis falhas.

O espaço inicial de observação é definido pelas propriedades do sistema no projeto. Os sensores e indicadores internos e externos coletam os dados que permitem a detecção das falhas e o processo de diagnóstico. O espaço de observação pode ser ampliado através de correlações e da fusão dos dados [Hamilton et al., 2001]. Para tanto, além de capacidade de processamento, é necessário o conhecimento das correlações existentes entre as várias fontes de informação.

A razão entre o espaço de diagnóstico e o espaço de observação normalmente é inferior a 1. Isto é devido à necessidade de informações redundantes para se efetuar

a detecção das falhas e a impossibilidade de associar um sensor com cada possível diagnóstico. Muitas técnicas diferentes são utilizadas para realizar o diagnóstico e tentar melhorar a razão entre o espaço de observação e o espaço de diagnóstico. Alguns métodos são muito utilizados, como as árvores de falhas [Visinsky, 1994], as redes neurais [Goel et al., 2000], os sistemas especialistas [Hamilton et al., 2001], o diagnóstico baseado em modelos(MBD), entre outros. De maneira geral a informação utilizada para o diagnóstico é dividida em cinco áreas principais [Hamilton et al., 2001]: projeto, sensores, histórico, missão e falhas.

O conhecimento proveniente do projeto é uma das informações mais importantes para a realização de diagnósticos. Esta inclui os diagramas dos circuitos existentes, o fluxo de informação, o projeto mecânico e os modelos funcionais; em suma, tudo que é capaz de descrever o sistema.

Uma das ferramentas mais utilizadas para conter as informações de projeto são árvores de falhas [Visinsky, 1994]. Estas explicitam as dependências entre os módulos de maneira hierárquica, permitindo a análise de confiabilidade de um sistema em função da confiabilidade dos subsistemas, e a correlação de várias falhas em função das dependências comuns.

O conhecimento do projeto pode conter, além das informações estruturais, os modelos funcionais (MBD) e neste caso permitir abordagens de diagnóstico baseadas em algum nível de simulação. Por exemplo, o conhecimento das equações lógicas de um circuito combinatório junto com vetores de entradas e saídas associados pode permitir um diagnóstico apurado. Em muitos casos esta abordagem pode ser totalmente inviável por restrições de processamento ou os modelos funcionais podem ser inadequados.

Outro conhecimento fundamental são as informações dinâmicas provenientes de indicadores e sensores do próprio sistema, correspondendo este ao espaço de observação direto. Os sensores são capazes de coletar dados sobre o próprio sistema e sobre o ambiente com qual interagem. Os indicadores podem ser considerados sensores de software que coletam os dados de execução, como alocação de memória, carga do processador, eventos do sistema operacional e outros.

O histórico de falhas e eventos de um sistema é também muito importante. Permite o ajuste das informações de confiabilidade de cada componente e do sistema completo em função das observações reais. Além disto, a análise de um histórico auxilia na detecção de pontos críticos e na apuração dos modelos de diagnóstico utilizados. Podem-se extrair também assinaturas de defeitos e correlações de eventos não percebidas durante o projeto. Pode-se dizer que os históricos ampliam o espaço de observação no tempo.

O conhecimento da missão ou da tarefa em execução permite a identificação dos recursos em uso e os disponíveis. Este conhecimento é importante porque o estado da missão pode alterar as probabilidades de falhas dos módulos, que dependem da

interação com outros componentes e com o ambiente. É muito mais provável que um módulo em uso apresente falhas antes que um módulo inativo, e que um módulo que permaneceu inativo por muito tempo apresente falhas logo que é ativado.

Essencial para o diagnóstico é saber quais são as falhas que foram detectadas no momento, ou seja, as informações incluindo as comparações e erros utilizados para identificar a presença de uma falha. Estas informações são a chave do processo de diagnóstico para a identificação da causa de uma falha.

As técnicas mais elaboradas de diagnóstico visam aumentar o espaço de diagnóstico, sem aumentar o espaço direto de observação. A utilização de históricos amplia de forma temporal o espaço de observação, enquanto os modelos funcionais também o ampliam através das simulações. Aumentar o espaço de diagnóstico com o uso de conhecimento continua a ser um grande desafio, pois as cinco fontes distintas de conhecimento são de natureza muito diferente, o que dificulta o uso simultâneo e a identificação de relações úteis para qualquer processo de diagnóstico, principalmente se existirem também restrições de tempo e de espaço para o processamento.

2.4 Tolerância a falhas adaptativa

Um sistema dotado de tolerância a falhas adaptativa possui a capacidade de ajustar dinamicamente a sua confiabilidade e seu desempenho devido a mudanças internas, externas ou alterações no seu objetivo instantâneo. Para atender este requisito, o sistema deve possuir um grande número de configurações diferentes e ser capaz de selecionar a mais apropriada para cada momento.

A tolerância a falhas tradicional envolve a definição do modelo de falhas e dos métodos de detecção e recuperação destas durante o projeto, juntamente com a alocação de todos os recursos necessários. Entretanto, a alocação estática de recursos tem se mostrado inapropriada para sistemas de tempo real, que operam em ambientes muito dinâmicos e variáveis [Gonzalez et al., 1997]. A tolerância a falhas adaptativa pode garantir a confiabilidade adequada de módulos críticos sobre restrições de recursos e restrições temporais, realizando a alocação da redundância disponível de forma dinâmica e se adaptando a estados sistêmicos e ambientais.

O propósito da tolerância a falhas adaptativa (*Adaptive Fault Tolerance - AFT*) é melhorar a confiabilidade, o desempenho e a capacidade de sobrevivência de um sistema através das seguintes características [Shokri et al., 1998, Shokri and Beltas, 2000]:

- Efetividade: A abordagem AFT deve aumentar de forma significava a confiabilidade de um sistema quando comparada ao mesmo sem redundância, mantendo os custos extras em limites aceitáveis.

- Melhorar a utilização dos recursos: O objetivo dos mecanismos reativos de adaptação é minimizar a utilização de recursos mantendo os requisitos de confiabilidade. Isto é realizado habilitando e desabilitando módulos de software e de hardware dinamicamente.

- Generalidade: A abordagem AFT deve ser geral o suficiente para tolerar uma variedade significativa de classes de falhas, tais como falhas de sensores, dos processadores, da memória, do sistema operacional e dos *softwares* aplicativos. Além de tolerar falhas transientes e permanentes.

- Resposta apropriada: Os mecanismos de AFT devem responder em um período de tempo aceitável a alterações no ambiente, no estado do sistema e a alterações na missão.

- Capacidade de configuração: O sistema pode receber parâmetros específicos da aplicação ou a da missão que afetam as decisões de adaptação durante a execução.

- Versatilidade: as políticas de adaptação devem considerar várias estratégias em resposta a comportamentos inesperados ou anômalos. Os parâmetros de recuperação de falhas podem ser modificados ou os atributos dos serviços redefinidos.

- Simplicidade: A interface entre os mecanismos de AFT e a aplicação original deve ser simples.

Os métodos empregados na AFT para efetivar a tolerância as falhas são os mesmos tradicionais, como a redundância modular ou uso de "*checkpoints*" e "*roolbacks*" e outros. A essência básica é selecionar o método adequado para um determinado módulo em função das restrições específicas. O mesmo módulo pode ter sua tolerância implementada de formas diferentes. A Figura 2.1 mostra algumas opções básicas de recuperação para um bloco de software sujeito a defeitos de programação ou falhas de processamento.

Neste exemplo, todo módulo necessita um teste de aceitação (**TA**) que detecta a presença de falhas [Bagchi, 2001, Randell and Xu, 1995]. A confiabilidade do sistema está relacionada diretamente à qualidade dos testes de aceitação. Este teste pode ser implementado através de mecanismos de votação, invariantes e premissas do módulo, ou utilizando assinaturas geradas durante a compilação ou a execução do mesmo. Os mecanismos de votação normalmente são os mais lentos, pois envolvem processos de comunicação, o que pode restringir o seu uso.

Quando as restrições de tempo de um módulo não são muito severas, o bloco pode ser executado novamente, no mesmo processador ou em outro diferente como na

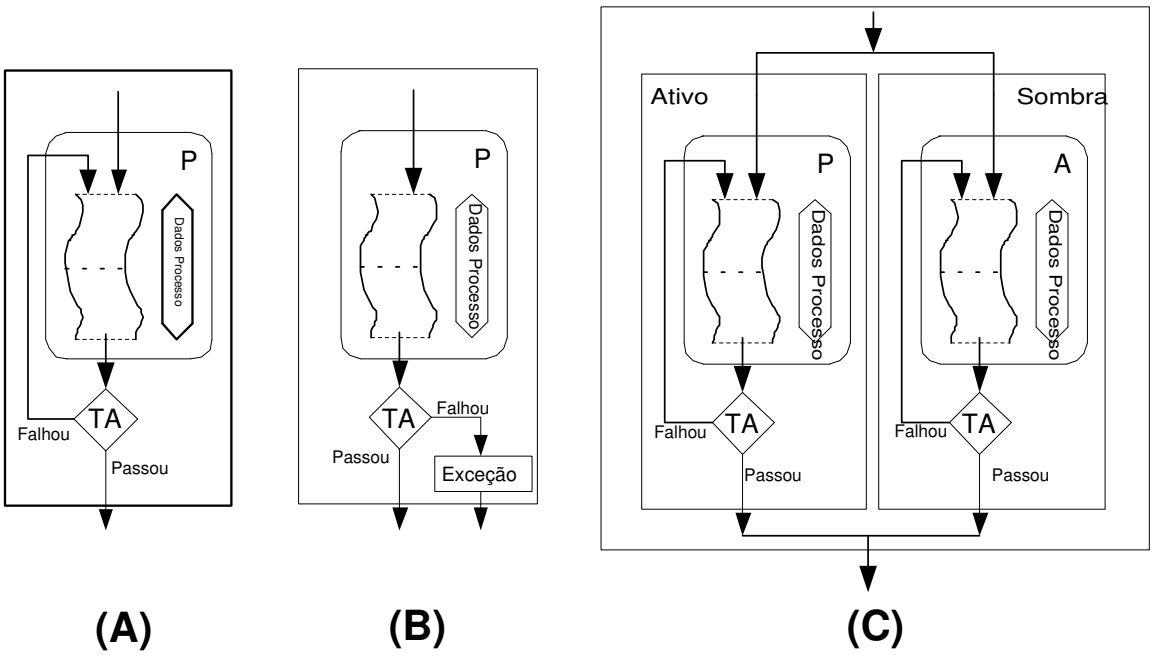


Figura 2.1: Modelos de redundância de processos utilizados para AFT.

Figura 2.1, ítem (A). Em alguns casos, ativar um módulo ou caminho alternativo na forma de uma exceção (ítem B) sem executar novamente o processo que apresentou a falha pode ser a recuperação mais adequada.

Em sistemas de tempo real nos quais os limites de tempo são críticos e é necessário garantir o tempo de resposta, são utilizadas cópias idênticas do módulo executando simultaneamente em dois ou mais processadores(ítem C). Caso a cópia principal falhe a secundária entra em ação quase instantaneamente.

Estas restrições e soluções já são utilizadas em sistemas de tempo real com tolerância a falhas tradicional, sendo que o método utilizado para cada processo é definido durante o desenvolvimento do projeto. O sistema final é dimensionado levando em conta sempre o pior caso de funcionamento.

A diferença básica da AFT, apresentada na Figura 2.2, é a possibilidade de troca do método dinamicamente, em função das mudanças de fatores internos e externos. As políticas de redundância de software utilizadas são definidas durante o projeto, mas a seleção da mais apropriada só é feita durante o funcionamento normal. Esta adequação permite uma confiabilidade muito maior em função dos recursos empregados. No caso de sistemas autônomos como naves, robôs, satélites e aviões onde a disponibilidade de recursos é um fator extremo e os objetivos se alteram ao longo do tempo, a AFT se mostra como a opção mais adequada. Os fatores que influenciam na escolha dinâmica das políticas são mostrados na Tabela 2.1 extraída de

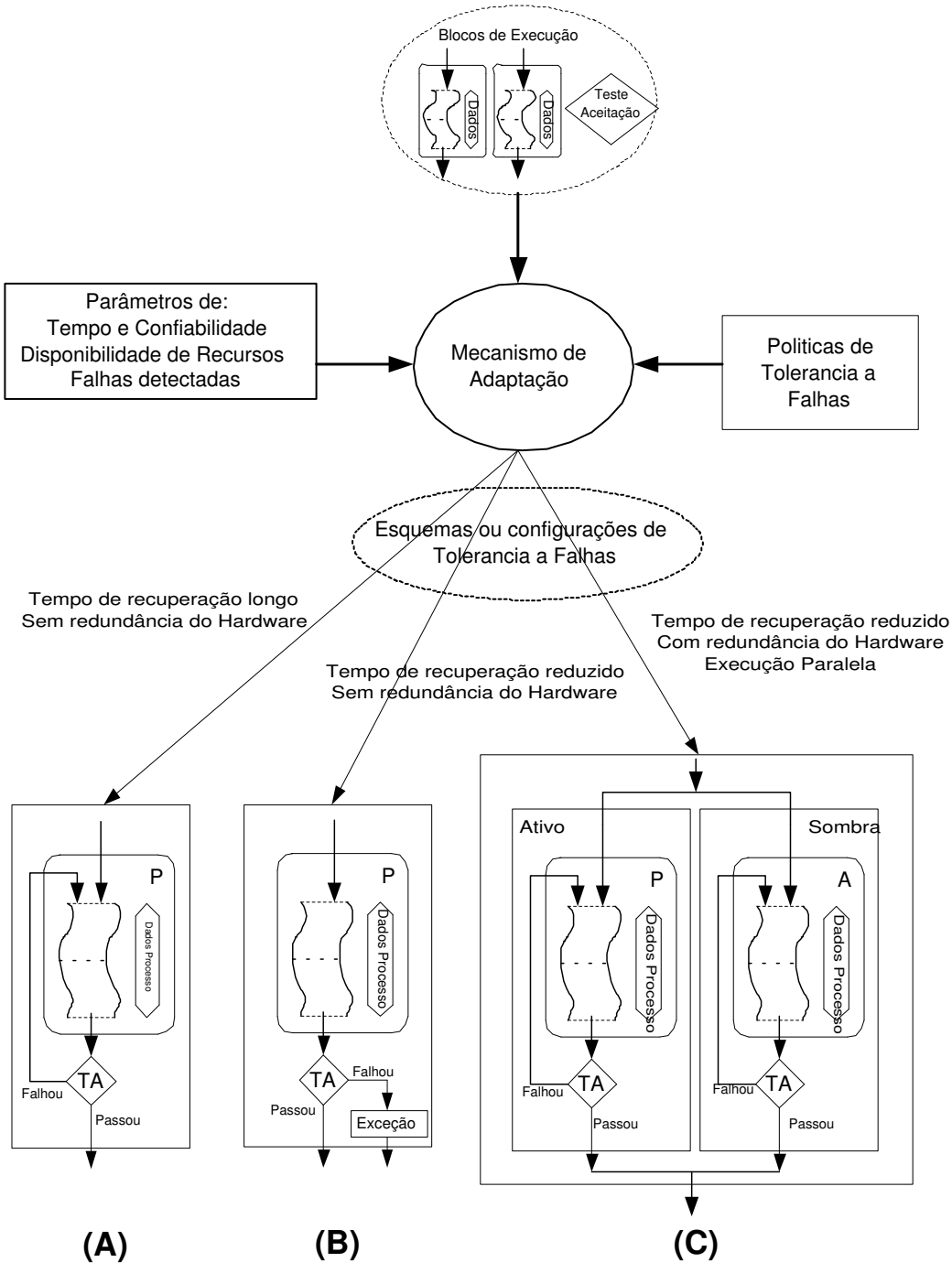


Figura 2.2: Mecanismo de adaptação às políticas de tolerância a falhas.

Classes	Exemplos
Ambientais	Mudanças de temperatura bruscas ou cíclicas. Regiões com alta radiação.
Fase da missão	Alterações no tempo de resposta. Recursos ou módulos necessários para a fase, incluindo sensores, atuadores e processamento.
Estado do Sistema	Autonomia disponível; falhas de sensores ou atuadores; mudanças de configuração do processador ou controle; falhas transientes ou permanentes dos módulos de processamento.
Perfis de usuários	Redefinição de tarefas ou objetivos; alteração da missão.

Tabela 2.1: Classes de mudanças potenciais que o mecanismo de adaptação deve responder.

[Hecht et al., 2000].

Nos trabalhos de Hechet et al [Hecht et al., 2000, Shokri and Beltas, 2000] foi criada uma camada intermediária de software entre a aplicação e o sistema operacional (VxWorks) para se implementar sistemas com AFT. Esta camada intermediária manipula todo o conhecimento necessário para a adaptação dos métodos de tolerância, permitindo que um módulo interno de decisão selecione as políticas adequadas a cada momento. Além disto, fornece para a aplicação uma camada de serviço com o controle de processos e com a comunicação confiável na forma de mensagens enviadas por canais lógicos.

No trabalho de Fohler [Fohler, 1997] a AFT foi implementada utilizando um escalonamento estático em sistemas de tempo real. O método implementado incorpora, dinamicamente durante a execução, as tarefas relativas à tolerância. Isto é realizado através da movimentação dos "*slots*" definidos no escalonamento estático, sem prejudicar as restrições de tempo previamente definidas. Hayes em [Kandasamy and Hayes, 1998] teve uma abordagem semelhante para tolerar falhas transientes em sistemas de tempo real embutidos, utilizando também um escalonamento estático. O sistema possui uma tabela contendo um conjunto de escalonamentos previamente calculados e a AFT é obtida através da seleção em tempo real da opção mais adequada à situação atual.

Concluindo, a tolerância a falhas adaptativa permite que o sistema concentre a utilização dos seus recursos nos elementos mais críticos para o sucesso da missão ou tarefa corrente. Portanto, a AFT é adequada para sistemas de tempo real multiprocessados ou não, mas que possuem fortes restrições na disponibilidade e utilização de

seus de recursos, além da necessidade de alta confiabilidade.

Os trabalhos sobre AFT encontrados concentram-se em sistemas de tempo real, tolerando essencialmente falhas de processamento (processador e memória) ou de codificação. As políticas de redundância utilizadas estão baseadas no uso de blocos de código totalmente equivalentes e facilmente intercambiáveis.

Quando o modelo de falhas considerado fica mais complexo, por exemplo, quando se inclui falhas em elementos perceptivos ou eletromecânicos, a redundância através de blocos equivalentes de software não é mais eficaz, pois garante a confiabilidade de processamento de informações incorretas ou a execução de ações infrutíferas. Neste caso, novas fontes de informações ou novas maneiras de realizar uma tarefa devem fazer parte das políticas de redundância disponíveis. O trabalho desenvolvido nesta tese buscou ampliar as possibilidades na criação das políticas de redundância de forma complementar às já existentes, como pode ser visto na Seção 5.1.

Capítulo 3

Introdução a sistemas de robôs

> Tudo o que um homem pode imaginar, outros homens podem realizar.
>
> Julio Verne (1828 - 1908)

A tolerância a falhas em robôs é uma área muito ampla, devido às características específicas dos robôs, as suas inúmeras aplicações e a grande variedade e diversidade de abordagens existente; soma-se a isto, a própria complexidade da tolerância a falhas. Um ponto muito importante é definir o que é um robô. No seu livro, Ronald C. Arkin [Arkin, 1998], apresenta as seguintes definições:

- *"Robotics Industry Association (RIA)" "a robot is a re-programmable, multi-functional, manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks"* ***[Jablonski and Posey 1985]***.
- *"the intelligent connection of perception to action"* ***[Brady 1985]***.
- *"An intelligent robot is a machine able to extract information from its environment and use knowledge about its world to move safely in a meaningful and purposive manner"* ***[Arkin 1998]***.

Segundo estas definições, pode-se dizer que um robô é praticamente qualquer tipo de sistema que interage com o meio através de sensores e manipuladores, realizando determinadas tarefas programadas. Neste trabalho, um robô é definido com um sistema dotado de um corpo físico, que possui um conjunto de sensores e atuadores, e um controle próprio capaz de definir as ações dos atuadores em função do seu estado interno e dos dados sensoriais. Para compreender melhor os problemas e as soluções

de tolerância a falhas utilizadas nos robôs, deve-se conhecer determinadas particularidades e propriedades dos sensores, atuadores e do controle. Estas particularidades serão descritas ao longo deste capítulo.

3.1 Atuadores

Os atuadores são quaisquer elementos capazes de provocar alguma alteração física no ambiente ou internamente ao robô. O motor conectado a um roda, um alto falante ou um farol podem ser considerados atuadores. Os atuadores mais comuns são motores, e geralmente estão relacionados à movimentação ou a manipulação de objetos.

Uma estrutura muito comum e conhecida nos robôs são os manipuladores, correspondendo aos braços mecânicos e garras, utilizados para manipular objetos e ferramentas. A construção dos manipuladores é realizada através da composição de seguimentos rígidos conectados através de juntas. Estas podem ser prismáticas ou rotacionais e tipicamente se movem em torno de um eixo. Cada movimento possível determina um grau de liberdade (DOF - *degree of freedom*). O movimento de cada junta é realizado através de um atuador, que pode estar conectado diretamente ou através de outros mecanismos como engrenagens, correntes.

Os atuadores também são responsáveis pela movimentação dos robôs, podendo ser de muitas formas: rodas, esteiras, hélices, pernas, etc. A forma de movimentação é muito importante, pois determina a capacidade de acesso do robô a diferentes ambientes.

Os atuadores, assim como quaisquer sistemas mecânicos, são sujeitos a erros e incertezas. Por mais preciso que seja o controle do motor de um determinado atuador, sempre vão existir diferenças entre o comando recebido e a movimentação realmente efetivada. Como é impossível eliminar totalmente estas diferenças, o controle de atuadores deve sempre trabalhar com um grau de incerteza.

3.2 Sensores

Os sensores são quaisquer elementos capazes de detectar uma informação do ambiente ou um estado interno ao robô e transformá-lo em um dado processável. São os elementos de hardware responsáveis por toda a percepção do ambiente e do próprio robô. Existe uma infinidade de sensores dos mais variados tipos trabalhando com informações de natureza diversa, como por exemplo: torque, pressão, distância, luminosidade, inclinação, posição e velocidade de uma junta e muitos outros. Os sensores são as portas de entradas de todas as informações do ambiente ou informações internas.

Os sensores são responsáveis tanto pela localização de marcos e obstáculos, quanto pelo conhecimento da posição de um manipulador. Assim sendo, a seleção do conjunto de sensores é parte fundamental no projeto de um robô, pois determina sua interação com o meio, e deve sempre fornecer informações suficientes para a execução das tarefas designadas.

Da mesma maneira que os atuadores, os sensores interagem com o ambiente real e estão também sujeitos a erros e incertezas. Objetivando uma melhor qualidade e precisão das informações percebidas, é comum utilizar-se de um conjunto de sensores com informações muitas vezes redundantes ou complementares. O processamento conjunto das informações provenientes de vários sensores para se obter uma melhor percepção do ambiente é conhecido como fusão de sensores [Murphy, 1994]. Existem várias técnicas diferentes de fusão sensorial, sendo que a qualidade dos resultados obtidos depende de vários fatores: conhecimento apurado das características e propriedades dos sensores; conhecimento da correlação entre as diversas fontes de dados, muitas vezes de naturezas diferentes; capacidade de processamento e tempo disponível para processar as informações.

Algumas vezes, informações sobre o ambiente ou sobre o próprio robô podem ser provenientes de sensores externos ou de outros integrantes de times cooperativos. Podem ser utilizados recursos de comunicação específicos como redes sem fio, infravermelho ou outros. É importante ressaltar que as informações externas também são sujeitas a erros e incertezas, podendo ser incluídas nos processos de fusão sensorial.

3.3 Controle

O controle de um robô é responsável por receber e processar os dados provenientes dos sensores, decidir a próxima ação e por enviar os comandos aos atuadores. Os comandos são definidos em função das informações provenientes dos sensores e do estado interno armazenado no controle. O robô percebe o ambiente, decide a ação a executar e age. A necessidade de um robô móvel reagir prontamente às alterações internas ou do ambiente estabelece rígidos limites para os tempos de resposta e muitas vezes impossibilita o uso de métodos ou algoritmos que necessitam de processamento intensivo. A decisão de desviar de um obstáculo após a colisão com o mesmo, no mínimo é inútil.

Existem grandes divisões no projeto e implementação do controle de robôs fixos e de robôs capazes de navegar no mundo real [Brooks, 1986]. Duas das linhas principais para abordagens de controle para sistemas móveis se iniciaram com os experimentos de Walter [Walter, 1950] e os trabalhos de Nilsson [Nilsson, 1969]. O modelo de Walter é conhecido como reativo enquanto o modelo de Nilsson é conhecido como deliberativo. Os robôs de Walter eram extremamente simples com o controle baseado em ações reativas ou reflexivas. Essencialmente, conjuntos de reflexos ou ações

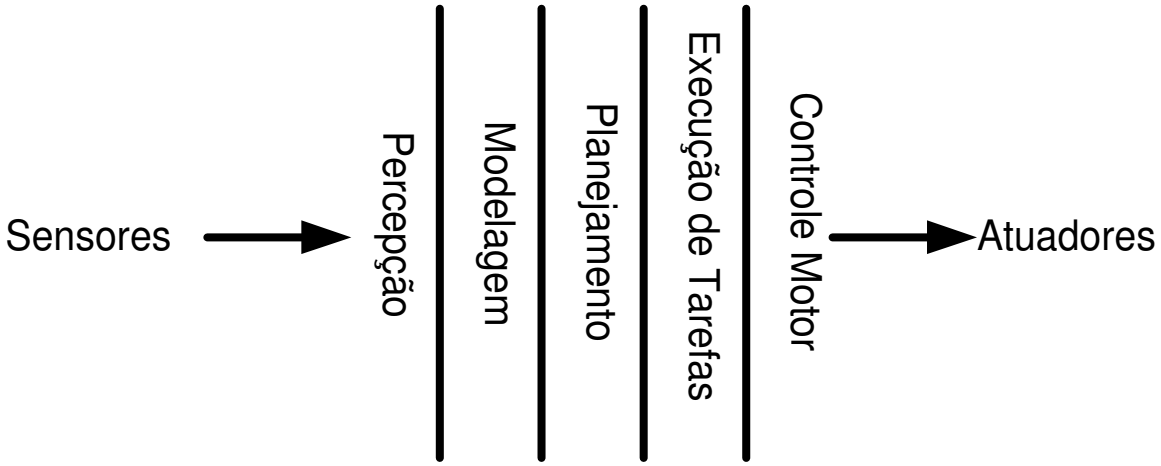


Figura 3.1: Modelo de controle deliberativo.

são associados diretamente a determinadas entradas perceptivas. Já o trabalho de Nilsson, descreve um robô muito elaborado conectado a um grande *Mainframe*. As entradas dos sensores eram processadas ou fundidas criando um modelo do mundo. Este modelo era analisado para, juntamente com os objetivos, criar um longo plano de ações. A execução deste plano deveria levar à conclusão do objetivo. Esta seqüência de processamento entre a percepção e ação é mostrada na Figura 3.1.

O modelo de Walter foi esquecido por muito tempo até que Brooks [Brooks, 1986] revigorou a abordagem com o controle baseado em comportamentos. A essência deste é aproximar a ação e a percepção sem utilizar um modelo de mundo que pode não corresponder com precisão à realidade. O controle é dividido em vários comportamentos independentes que conectam diretamente a percepção a ação. As ações são produzidas pela interação e concorrência entre os diversos comportamentos, como mostrado na Figura 3.2. Alguns pontos são considerados chave para o projeto baseado em comportamentos:

- Manter a conexão entre a percepção e a ação mais próxima e simples possível. Conseqüentemente, mantendo a eficiência e a velocidade da reação.
- Minimizar a interação entre os diversos circuitos de controle, favorecendo assim a independência dos comportamentos.
- A possibilidade de se incluir no projeto de um robô um novo circuito ou lógica específica que vai atuar em uma determinada situação, evoluindo assim as capacidades globais, sem a necessidade de alterar os circuitos já existentes.
- A estruturação do controle em diversos níveis que permite a inclusão de novos circuitos e a composição de novas camadas que utilizam as camadas inferiores.

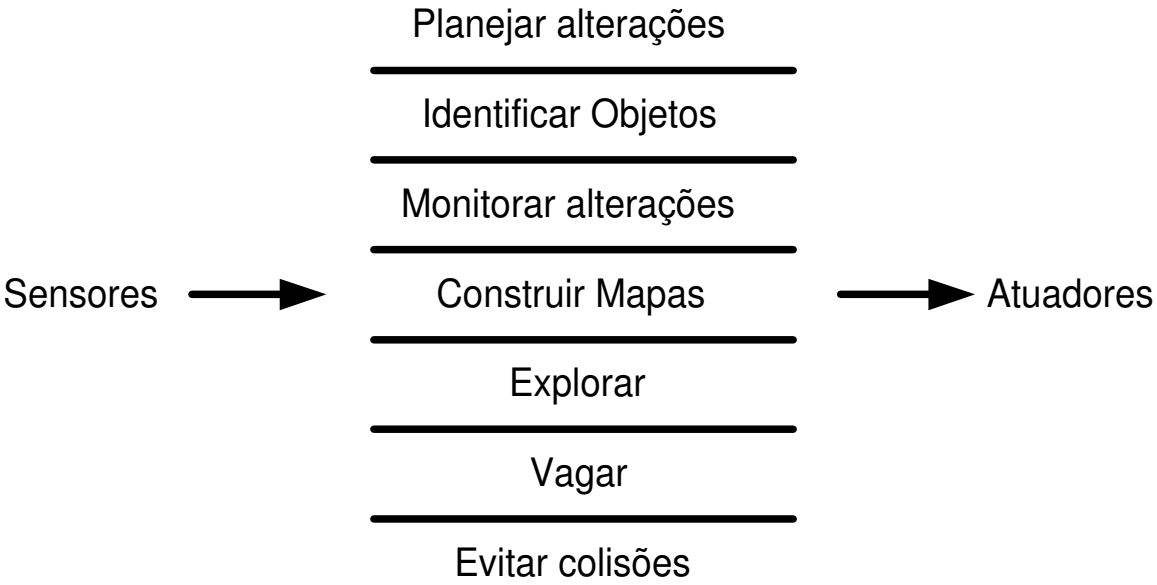


Figura 3.2: Modelo de controle baseado em comportamentos.

> A inteligência é ampliada sem aumentar a complexidade de cada circuito ou comportamento individualmente.

Os detalhes completos do modelo de controle baseado em comportamentos foram publicados por Brooks [Brooks, 1989a, Brooks, 1989b, Brooks, 1999]. Foi utilizado um robô de seis pernas, o Genghis. Os princípios desta abordagem baseada em comportamentos para controle de robôs continuaram a ser desenvolvidos com Patti Maes e outros [Maes, 1990], Brooks [Brooks, 1991c, Brooks, 1991b, Brooks, 1991a, Brooks, 1985], Arkin [Arkin, 1989, Arkin, 1998] e Mataric [Mataric, 1992a, Mataric, 1994, Weber et al., 2000, Goldberg and Mataric, 2000] e outros. Algumas diferenças entre os trabalhos serão apresentadas na próxima seção.

No modelo deliberativo as respostas comportamentais do robô emergem da interação entre os objetivos da missão, o plano de ação elaborado e o modelo de mundo que foi construído utilizando os dados sensoriais. Nos sistemas baseados em comportamento, as respostas comportamentais são explicitamente programadas, associando da forma mais direta possível os estímulos sensoriais com ações. Neste caso, os objetivos e planos não são expressos explicitamente, mas implicitamente através da interação entre os comportamentos.

Cada uma das abordagens tem suas vantagens e desvantagens. O modelo deliberativo realiza suas decisões utilizando um modelo abstrato do mundo em que o robô opera. A precisão do modelo do mundo está relacionada com a capacidade sensorial e com o processamento utilizado na sua criação. Se o modelo não é apurado, o robô

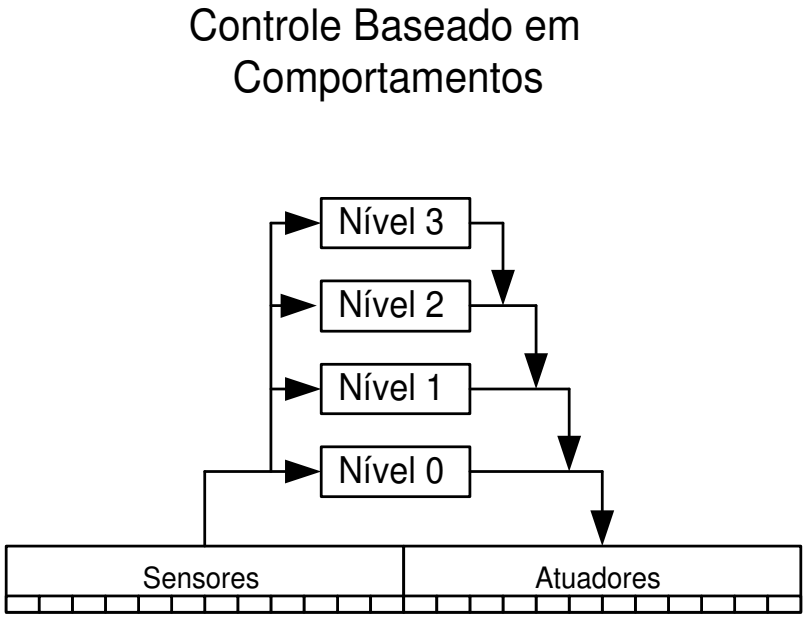


Figura 3.3: Representação gráfica do modelo de controle baseado em comportamentos.

pode não decidir adequadamente suas ações. Por isso, o controle deliberativo muitas vezes não se apresenta adequado a ambientes variados e muito dinâmicos, pois a atualização do modelo pode não ser realizada em velocidade suficiente para reagir apropriadamente às alterações ambientais. Quando o modelo de representação é apurado o suficiente para uma determinada tarefa, a abordagem deliberativa apresenta bons resultados, pois permite assim realizar um planejamento e otimizações na seleção e execução das ações.

Os modelos baseados em comportamentos utilizam a informação sensorial em formas mais primitivas, evitando a utilização de métodos complexos para realizar a fusão dos dados provenientes dos sensores, como mostrado na Figura 3.3. A informação sensorial pode ser utilizada por vários comportamentos distintos de forma independente, permitindo um ciclo de reação entre os dados recebidos dos sensores e a ação, curto e eficiente. Desta forma, o modelo baseado em comportamento apresenta bons resultados em ambientes variáveis e dinâmicos. Entretanto, a característica do controle ser dividido em muitos blocos independentes (comportamentos), dificulta qualquer processo de planejamento ou otimização global.

3.4 Arquitetura baseada em comportamentos

O controle baseado em comportamentos apresenta várias linhas diferentes. Cada uma das abordagens varia principalmente em função dos métodos de ativação dos comportamentos e na arbitragem das saídas (*Action Selection*)[Bryson, 2000].

O modelo de controle original de Brooks [Brooks, 1986] é conhecido com "*subsumption*". O controlador é construído em termos de níveis de competência. Cada nível oferece uma base para o nível superior. A 'inteligência" ou capacidade do sistema é ampliada a cada novo comportamento inserido. A implementação original de Brooks [Brooks, 1989b] é baseada em um conjunto de máquinas de estados finitos ou FSA (Finite State Acceptor) que interagem entre si. Devido à dificuldade de programação utilizando diretamente as FSA, foi desenvolvida uma linguagem chamada de "*Behavior Language*" [Brooks, 1990], que oferece elementos abstratos que são compilados para um conjunto de FSA executável. Continuando o desenvolvimento, Brooks [Brooks, 1991d] explicita os seguintes conceitos:

- **Situado**[1]: define a habilidade dos robôs em sentir o ambiente a sua volta evitando o uso de representações abstratas.
- **Personificação**[2]: define os robôs como criaturas físicas que devem experimentar o mundo diretamente e não através de simulação.

O desenvolvimento do controle baseado em comportamentos continua com Arkin, que o combina com a teoria de esquemas (*Schema Theory*) desenvolvida por Arbib [Arbib, 1992]. O conceito biológico de esquemas Motores (*Motor schemas*) é então aplicado ao controle de robos [Arkin, 1989, Arkin, 1995, Arkin, 1998]. Os esquemas perceptivos são embutidos nos esquemas motores, de forma a reagir a estímulos tão rápido quando possível. Os esquemas perceptivos podem ser definidos recursivamente tornando-os capazes de extrair informações mais elaboradas e significativas para ativar apropriadamente os esquemas motores.

Patti Maes [Maes, 1989b, Maes, 1989a, Maes, 1990, Maes and Brooks, 1990] publicou um método baseado na seleção de ações através do espalhamento de ativações geradas por objetivos e por módulos que detectam determinadas situações pré-definidas.

Mataric [Mataric, 1992a, Mataric, 1992b] desenvolve uma heurística para desenvolvimento de controles comportamentais. Os comportamentos são definidos em um nível mais alto e refinados seguidamente até que possam ser aterrados ou fundamentados nos dados sensoriais.

[1] *Situatedness*
[2] *Embodiment*

No intuito de desenvolver tarefas mais complexas, Firby e Slack [Firby, 1994, Firby et al., 1995] desenvolveram um trabalho, no qual um conjunto de tarefas é especificado por seqüências de ações, que ativam um conjunto de habilidades específicas (*skill*), que se assemelham a comportamentos. Este trabalho não é considerado puramente comportamental porque utiliza módulos de resolução de problemas para realizar planejamento, que define a seqüência de ações apropriadas para cada tarefa.

Monica Nicolescu e Mataric [Nicolescu and Mataric, 2000a, Nicolescu and Mataric, 2000b] ampliaram o modelo comportamental definindo o conceito de comportamentos abstratos. Seqüências de comportamentos abstratos especificam a realização de um determinado objetivo ou tarefa. Os comportamentos abstratos são interconectados com comportamentos primitivos, possibilitando uma interação através da distribuição de ativação e inibições. A estrutura proposta permite o desenvolvimento de tarefas complexas utilizando o mesmo conjunto de comportamentos primitivos.

O modelo de Brooks que define um paradigma conhecido por PAB (*Port-Arbitrated behavior Paradigm*) foi generalizado para sistemas multi-agentes por Werger [Werger, 2000]. Neste trabalho foi definida a linguagem Ayllu que facilita a implementação do controle comportamental de Brooks em um time de robôs interconectados através de rede IP.

Todas as abordagens baseadas no modelo comportamental foram inspiradas nos comportamentos animais e no conhecimento sobre processos biológicos de percepção, cognição e ação. Implementar estes processos em qualquer tipo de sistema não é uma tarefa simples e abre possibilidades para várias e diferentes abordagens, cada uma com propriedades próprias, vantagens e desvantagens. Podem-se destacar duas divisões principais entre as abordagens: a forma de saída ou resposta dos comportamentos e a coordenação dos comportamentos.

Os comportamentos devem enviar comandos para os atuadores ou motores definindo a força, velocidade e direção destes. Os comandos podem ser discretos ou contínuos. Os discretos correspondem a um conjunto finito de valores pré-determinados. O controle de velocidade de um motor pode receber os comandos "frente", "trás" e "parado", cada um correspondendo a um valor real de velocidade ou força pré-determinados. Os comandos contínuos podem assumir qualquer valor real dentro de uma faixa pré-determinada de atuação.

Além do tipo de resposta fornecida para comandar atuadores, outra característica marcante é o paralelismo inerente à ativação dos comportamentos. É possível se ter vários comportamentos ativos no mesmo instante e fica clara a necessidade de selecionar a melhor saída do sistema. A seleção, também chamada de coordenação de comportamentos, pode ser realizada de diversas formas e deu origem a diferentes abordagens do controle baseado em comportamento. A coordenação de comportamentos pode ser realizada em duas formas básicas: métodos competitivos e métodos

cooperativos.

Os métodos de controle competitivos garantem que apenas a saída de um comportamento é utilizada para o controle dos atuadores. No modelo de Books conhecido como "*subsumption´´*, vários comportamentos são ativados simultaneamente e a saída é selecionada através de critérios de supressão. Existe uma prioridade previamente definida entre todos os comportamentos durante o projeto.

Patti Maes [Maes, 1990, Maes and Brooks, 1990] publicou um método baseado na seleção de ações. É proposta uma rede de comportamentos conectados com objetivos e sensores. A execução dos comportamentos é realizada quando a ativação recebida de objetivos e dos sensores ultrapassa um determinado limite("*threshold*"). Nesta abordagem, apenas um comportamento é ativado de cada vez, não existindo uma prioridade explícita entre eles e nem a necessidade de coordenar as saídas. A ativação é controlada por um conjunto de parâmetros existentes nos próprios comportamentos, nos sensores, nos objetivos e nas interconexões existentes. Uma vantagem desta abordagem é a possibilidade de alterar os parâmetros dinamicamente, permitindo adaptações durante a execução.

Quando se utilizam métodos baseados em prioridade, com supressão de saídas para arbitrar as respostas dos comportamentos, a informação contida nos comandos suprimidos é totalmente ignorada. Em muitos casos, pode não ser o ideal. Imagine um autômato que está seguindo um alvo e precisa desviar de um obstáculo. Para o comportamento de desvio pode não importar se vai virar para a direita ou esquerda, entretanto a direção escolhida pode ser decisiva para se alcançar o alvo. O comportamento de desvio deve portando, ou receber informações sobre a posição do alvo, ou permitir que suas saídas sejam combinadas ao comportamento que segue o alvo.

Os métodos que combinam saídas são conhecidos como coordenação cooperativa. Nestes métodos, as respostas dos comportamentos são somadas ou fundidas através de alguma função previamente definida. Ou seja, os comandos enviados aos atuadores são dependentes de todos os comportamentos ativos simultaneamente. Não existe, portanto, uma prioridade explicitamente definida em relação aos comportamentos ativos. Existem muitas maneiras para realizar a fusão de comandos tanto para comandos discretos, quando contínuos [Bryson, 2000]. Uma das maneiras mais conhecidas, utilizada nos esquemas motores por Arkin [Arkin, 1998], é a representação da percepção externa através de campos potenciais de atração e repulsão. O objetivo atrai o robô enquanto os obstáculos o repelem. A soma destes resultados determina a trajetória final. Os resultados dos comportamentos ativos simultaneamente são combinados ou fundidos através de soma vetorial.

Em alguns casos a fusão de comandos realizada apenas com a soma ou combinação de saídas pode também apresentar problemas. Se um comportamento comanda para virar a direita (10^O) e outro diferente comanda para virar a esquerda (10^O) o comando resultante da soma será (0^O), comandando o robô para continuar seguindo na mesma

direção, o que pode ser um problema.

Em outras palavras, a utilização das prioridades de forma explícita como no modelo de "subsumption" de Brooks pode levar a perda de informações produzidas pelos comportamentos suprimidos, e a fusão de comandos pode levar a ações inadequadas ou imprevisíveis. É claro que os exemplos apresentados podem ser facilmente resolvidos se a funcionalidade dos comportamentos for ampliada aumentando as informações utilizadas como estímulos ou a comunicação entre eles.

Entre os princípios do controle comportamental estão a simplicidade e independência no desenvolvimento dos comportamentos de cada nível. Mantendo estes princípios, outros métodos de arbitragem foram desenvolvidos, combinando a competição com a cooperação entre as respostas de atuação.

Um método baseado em votação conhecido como DAMN (Distribuited Architecture for Mobile Navigation) foi desenvolvido por Rosenblatt [Rosenblatt, 1997]. Cada comportamento do sistema em vez de escolher um comando específico, vota em um conjunto predefinido de comandos discretos para os atuadores, possuindo um número de votos. A coordenação é realizada se contabilizando os votos e selecionando o comando vencedor, e este é efetivamente executado. Neste caso, também não há uma prioridade explícita entre os diversos comportamentos. Este método é considerado competitivo, mas apresenta um nível de cooperação. A importância de um comportamento pode ser alterada quando se modifica o número de votos distribuídos no sistema.

Uma outra abordagem para fusão de atuação muito interessante é encontrada em Payton et al [Payton et al., 1992]. Cada comportamento pode responder três tipos de valores possíveis para um comando.

Faixa (*Zone*): Define um limite inferior e superior para um comando.

Limite (*Clamp*) : Define um limite inferior ou um superior para um comando.

Preciso (*Spike*) : Define um valor específico e único para o comando. Neste caso é utilizado um controle de prioridade usual.

Os comandos gerados pelos comportamentos podem então ser descritos como variáveis de controle e não saídas diretas. O processo de fusão dos comandos é realizado no intuito de se atender às restrições de todos os comportamentos simultaneamente. Caso não seja possível, são utilizados critérios de prioridade definidos pelo estado do sistema. A vantagem desta representação é a possibilidade de que cada comportamento pode criar aproximações inteligentes e constantes para sua função contínua de preferência.

Existem ainda outras abordagens para os mecanismos de coordenação de comportamentos cooperativa, competitiva ou híbrida. Pirjanian [Pirjanian, 1999], fez um

apanhado geral sobre os métodos de coordenação de comportamentos utilizados. Joana Bryson [Bryson, 2000] estudou os mecanismos de seleção da ação correlacionando-os com hipóteses de psicologia.

Firby et al [Firby et al., 1995] desenvolveram um modelo híbrido dividido em duas camadas interconectadas. Um planejador que utiliza uma biblioteca de tarefas organizadas na forma de árvores, que em função das tarefas ativas, um conjunto de habilidades do nível inferior são habilitadas. As habilidades perceptivas interagem com habilidades de ação permitindo um controle eficiente. O planejador interage com as habilidades perceptivas recebendo estímulos que vão permitir a seleção de novas tarefas.

Ainda existem muitos outros trabalhos desenvolvidos na área, mas estes exemplos são significativos nas tecnologias utilizadas.

3.5 Controle de arquitetura híbrida

Buscando o melhor de cada modelo, muitas abordagens com arquiteturas híbridas foram desenvolvidas. O modelo comportamental é utilizado nos níveis mais básicos de controle e o modelo deliberativo é utilizado no planejamento e controle de tarefas. O problema de projeto utilizando arquiteturas híbridas reside na divisão destas competências e na interação entre elas de forma adequada. A arquitetura híbrida é mostrada na Figura 3.4. O nível mais alto de controle define a cada momento as funções de controle ou comportamentos de mais baixo nível que devem estar ativos. Para tanto, o nível deliberativo tem acesso aos dados sensoriais e a outros indicadores de mais baixo nível.

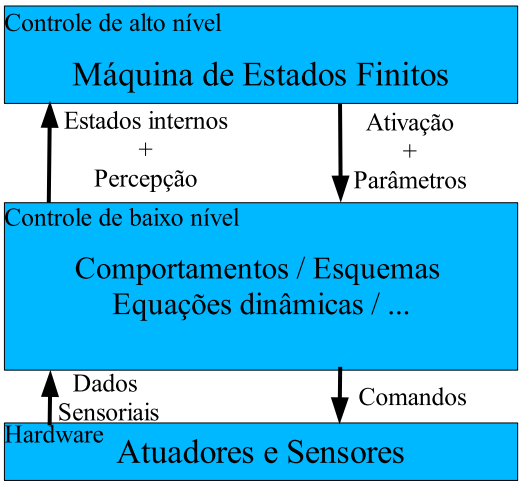


Figura 3.4: Arquitetura híbrida.

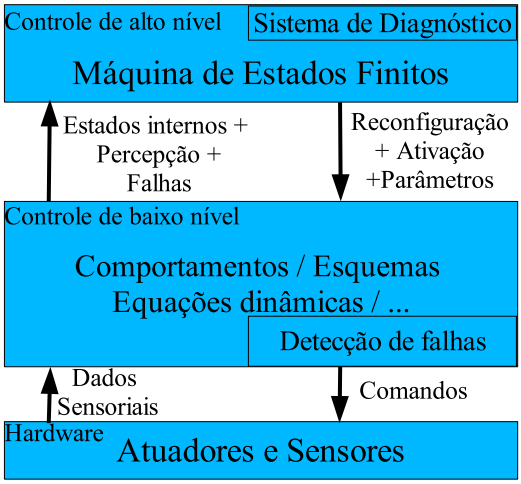


Figura 3.5: Arquitetura híbrida com tolerância a falhas.

Os robôs que mais necessitam de tolerância a falhas são projetados para trabalhar em ambientes hostis, imprevisíveis e de difícil modelagem. Esta característica favorece o uso de um controle comportamental. Quando se desenvolve um sistema tolerante a falhas deve-se considerar a possibilidade da presença de subsistemas defeituosos cuja influência deve ser isolada. A ativação ou desativação de módulos defeituosos em uma abordagem comportamental significa alterar a interação e as mensagens de controle entre os comportamentos, tornando o projeto muito mais complexo.

Para evitar o aumento de complexidade na ativação e inibição dos comportamentos para se reagir a falhas, foi escolhida neste trabalho a arquitetura híbrida. A abordagem utilizada, mostrada na Figura 3.5, integra no baixo nível as funções de detecção de falhas e ao controle de mais alto nível, a capacidade de recuperação de falhas utilizando o mecanismo comum de ativação de funções de mais baixo nível.

No trabalho desenvolvido nesta tese é importante ressaltar que as funções de controle de baixo nível não são restritas somente a comportamentos, podendo seguir outras abordagens ou variações. O nível inferior pode ser implementado por blocos de processamento interconectados, apropriados a abordagens tais como:

- Controle reativo (Walter 1950);
- Arquitetura de controle por comportamentos ([Brooks, 1999, Mataric, 1997]);
- Esquemas perceptivos e esquemas motores ([Arkin, 1998,

Murphy and Hershberger, 1996]);

- Controles com equações contínuas ([Chaimowicz et al., 2001]).

As abordagens de baixo nível podem variar, mas alguns dos problemas como fusão de comandos, arbitragens e outros destacados para a abordagem comportamental são problemas comuns a elas.

No nível mais alto de controle, a arquitetura híbrida utilizada nesta tese é implementada com uma máquina de estados finitos. Cada estado ativa um conjunto de funções de controle que interagem diretamente com os sensores e atuadores. O uso de uma máquina de estados finitos para este nível de controle apresentou algumas vantagens para o trabalho desenvolvido. A sua eficiência é decorrente da própria simplicidade e da facilidade para ser expandida e incluir novos estados adequados às adaptações ou detecção e recuperação de falhas.

Capítulo 4

Tolerância a falhas em robôs

> Se você rouba idéias de um autor, é plágio. Se você rouba de muitos autores, é pesquisa.
>
> Wilson Mizner (1876-1933)

Alguns aspectos construtivos dos robôs e do seu controle determinam particularidades bem específicas na implementação da tolerância a falhas. Como foi visto no capítulo 2, a tolerância pode ser dividida genericamente em detecção de falhas, diagnóstico e recuperação da falha. Neste capítulo, estes aspectos da tolerância a falhas são correlacionados com os elementos básicos dos robôs: controle, sensores, atuadores e qualquer outro elemento elétrico ou mecânico existente no sistema. A variedade destes elementos torna a natureza e a origem de suas falhas extremamente diversa, além de muitas vezes dependerem de condições ambientais sobre as quais não se tem controle, ou se tem uma percepção muito reduzida [Hamilton et al., 2001]. Outra possibilidade é considerar o sistema robótico tolerante a falhas como um time cooperativo e não apenas como um robô individual. Portanto, a tolerância a falhas em robótica envolve a combinação de todas estas questões. Neste capítulo alguns aspectos são considerados e discutidos, juntamente com algumas soluções encontradas na literatura que nortearam o desenvolvimento deste trabalho.

4.1 Detecção de falhas

Os robôs podem apresentar falhas em qualquer elemento construtivo. Como foi visto, toda a percepção do próprio robô e do ambiente é realizada pelos dados sensoriais, portanto estes dados, juntamente com o conhecimento de expectativas internas, são utilizados para detectar as falhas presentes nos próprios sensores e atuadores.

Enfatizando, a detecção de falhas sempre se baseia na existência de informações redundantes, as quais apresentam valores discrepantes. Se existem dois ou mais sensores que fornecem informações correlacionadas, a detecção pode ser realizada comparando os valores obtidos. A redundância pode existir mesmo que os sensores sejam diferentes. Por exemplo, em uma junta de um manipulador é possível se comparar a saída de um sensor de velocidade com a derivada de um sensor de posição, ou comparar a saída do sensor de posição com a integral do sensor de velocidade [Visinsky, 1994]. A comparação utilizada pode ser ainda mais complexa, como se conferir a distância de um objeto utilizando algum tipo de sonar e um sistema de visão [Murphy and Hershberger, 1999]. Fica claro que, mesmo existindo capacidade sensorial redundante, nem sempre a utilização é direta e pode exigir um grande processamento, incluindo a fusão dos dados de vários sensores.

O conhecimento da variação dos valores obtidos do sensor também pode ser utilizado na detecção de falhas. Suponha que os dados sucessivos de um sensor, obtidos em uma determinada taxa de amostragem, variam em torno de dez unidades. Se dois valores sucessivos apresentarem uma diferença de mil unidades, é uma indicação forte da presença de uma falha. O mesmo acontece quando o valor obtido de um sensor sai completamente de uma faixa aceitável ou previsível. Este tipo de detecção é muito utilizada para eliminar erros instantâneos ou espúrios nas leituras, o que acontece freqüentemente [Murphy, 1994]. A informação redundante utilizada é o conhecimento prévio do comportamento do sensor.

A presença de erros nos sensores exige que as comparações sempre sejam realizadas com aproximações. O conhecimento do funcionamento de um sensor e da modelagem da sua incerteza é essencial para apurar a detecção das falhas [Vemuri and Polycarpou, 1997]. Muitos métodos são utilizados para se detectar falhas como modelos analíticos e diferentes análises matriciais [Hamilton et al., 2001, Redimbo, 1998] e modelos de aproximação não lineares com parâmetros ajustáveis [Vemuri and Polycarpou, 1997].

Um dos métodos mais importantes é o filtro de Kalman [Kalman, 1960], utilizado tanto para melhorar a precisão dos valores de vários sensores, quanto para se detectar falhas através da análise dos resíduos [Redimbo, 1998]. Os trabalhos desenvolvidos por Roumeliotis e Sukhatme [Roumeliotis et al., 1998a, Roumeliotis et al., 1998b, Goel et al., 2000] utilizam múltiplos filtros simultaneamente para se detectar as falhas.

No controle comportamental, nas abordagens de Ferrell e Murphy [Ferrell, 1994, Murphy and Hershberger, 1996, Murphy and Hershberger, 1999] a detecção de falhas sensitivas é realizada por um monitor de consenso, o qual compara as saídas dos sensores com um equivalente lógico ou virtual.

Os sensores são sujeitos a variações de funcionamento ao longo da vida útil. Estas variações podem ser vistas como erros ou perda de calibração e podem ser corrigidas

alterando os parâmetros relacionados ao processamento da saída do sensor. O processo de recalibração é essencial para os robôs, pois evita a perda das informações de um sensor desnecessariamente. A necessidade de recalibração é detectada quando todas as leituras de um sensor apresentam erros com uma variação comum em relação ao esperado [Horswill, 1994].

O tratamento de falhas transientes ou temporárias também é importante. Os sensores são afetados tanto por condições internas quanto externas. Estudos mostram que as falhas transientes chegam a 22% em naves espaciais [Hecht et al., 2000], o que torna essencial, além do isolamento, a sua reintegração ao sistema. As falhas transientes podem ser provocadas por condições ambientais que inviabilizem o uso do sensor; por exemplo, a presença de radiação atrapalha sensores infravermelhos; neblina ou fumaça podem inviabilizar sistemas de visão.

Controle com métodos automáticos de recalibração de sensores e atuadores e o tratamento para falhas transientes são encontrados nos trabalhos de Payton [Payton et al., 1992], Ferrell [Ferrell, 1994] e Murphy [Murphy and Hershberger, 1999]. O trabalho de Murphy inclui ainda ações específicas para realizar o diagnóstico, que buscam aumentar o espaço de observação do sistema.

Os sistemas robóticos podem apresentar, além de falhas nos sensores, falhas nos atuadores e em outros elementos mecânicos. As falhas, neste caso, são detectáveis se existir algum sensor diretamente associado ao elemento mecânico, como acontece nas juntas, ou se na interação com o mundo é notado um comportamento indevido. Nos dois casos a detecção é realizada através da percepção do robô, ou seja, através dos dados sensoriais.

A detecção de falhas em atuadores é realizada através da comparação entre um estado previsto e o percebido. Suponha que o controlador de um robô seja capaz de perceber sem falhas a posição de uma junta, a qual está no instante t^0 na posição de 10^o. O controlador envia um comando para que no instante t^1 a junta esteja na posição de 20^o. O valor y^1 de posição percebido no instante t^1 é comparado com o valor esperado ($[y^1 - 20^o] = erro$). Se for maior que a incerteza natural no atuador é assumido que existe uma falha no atuador ou em algum outro elemento mecânico relacionado. Alguns pontos são importantes de se destacar utilizando ainda o mesmo exemplo da junta. Estes evidenciam o problema do diagnóstico que será tratado na próxima sessão.

- Se o defeito é em um motor ou em algum elemento de transmissão é praticamente impossível determinar, sem o uso de outras fontes de informação.

- A percepção na junta foi considerada sem falhas. No caso real, se existe apenas um sensor na junta, este é também sujeito a defeitos. Portanto, não é possível determinar se a falha foi no atuador ou no sensor.

- A falha também pode ser proveniente da interação com o ambiente. Por exemplo, o manipulador pode colidir com algum obstáculo e não alcançar a posição desejada.

Uma questão chave na detecção de falhas de atuadores é determinar os valores limites para os erros, distinguindo entre a operação normal e a presença de defeitos; se a faixa de limites for muito estreita, o sistema pode acusar vários erros falsos. Se for muito larga, vai demorar mais tempo para acusar a falha, podendo comprometer a segurança do ambiente ou a integridade do sistema. A determinação destes limites é realizada essencialmente através do desenvolvimento de um modelo de incerteza para o sistema, no qual os erros máximos nos atuadores e sensores são previstos, como pode ser visto no trabalho de Visinsky [Visinsky, 1994].

Os erros presentes nos sistemas mecânicos podem ser determinados por variações em escalas microscópicas. Um robô, como qualquer outro sistema mecânico, nunca é idêntico a outro nesta escala de detalhes, portanto o modelo de incertezas deve considerar estas diferenças. Os parâmetros utilizados na detecção de falhas devem ser ajustados a cada robô individualmente para se garantir a qualidade do processo.

4.2 Diagnóstico

A detecção de falhas é essencial, mas normalmente não é suficiente para a implementação eficaz da tolerância a falhas. A tolerância a falhas depende do universo possível de falhas, do espaço de observação das falhas, da redundância existente e da capacidade de selecionar a ação corretiva mais adequada, o que pode muitas vezes, estar totalmente dependente da identificação precisa do defeito [Murphy and Hershberger, 1999], tornando necessário realizar um processo de diagnóstico.

Quando não é possível a identificação correta do defeito, todos os elementos possíveis de causar a falha são colocados sob suspeita, deixando para o controle três opções básicas, sendo que a melhor depende da aplicação:

- Continuar utilizando os elementos suspeitos com o risco de erros, até que sejam coletadas informações suficientes para se realizar o diagnóstico.

- Considerar que todos elementos suspeitos estão defeituosos e perder funcionalidades e as capacidades de realizar determinadas tarefas associadas a eles.

- Entrar em um modo de diagnóstico, no qual o sistema procura coletar mais informações sobre os elementos suspeitos, executando ações com um risco reduzido ou controlado.

Mesmo com informações detalhadas de projeto, a identificação correta de um defeito pode ser muito difícil devido às múltiplas dependências entre os elementos constituintes do robô e a existência de um espaço de observação proporcionado pelos sensores muito reduzido em relação ao espaço de falhas [Hamilton et al., 2001], especialmente considerando os robôs autônomos que são sujeitos a grandes variações ambientais. Visando melhorar a qualidade do diagnóstico são utilizados vários outros métodos no intuito de identificar padrões de associação entre falhas observadas e defeitos reais. Pode-se exemplificar o uso de redes neurais ou sistemas especialistas capazes de criar correlações matemáticas dos dados.

Segundo Roumeliotis et al. [Goel et al., 2000] a detecção de falhas é relativamente simples e pode ser realizada através do uso de apenas um filtro de Kalman representando o modelo nominal do sistema. O problema crítico é identificar o que está acontecendo de errado, principalmente quando é necessário identificar com precisão as falhas mecânicas, falhas dos sensores e falhas devido às condições adversas do ambiente. Falhas dos atuadores e manipuladores são ambas detectadas através das informações provenientes dos sensores, e em muitos casos às assinaturas de falhas mecânicas são praticamente idênticas a assinaturas de falhas sensoriais.

Nos trabalhos de Roumeliotis [Roumeliotis et al., 1998a, Roumeliotis et al., 1998b] foram utilizados bancos de filtros de Kalman. Cada filtro assume que um tipo diferente de falha ocorreu e utiliza o modelo do sistema e sensores adequado para prever o comportamento do robô. No trabalho de Goel e Roumeliotis [Goel et al., 2000] os resíduos de cada filtro de Kalman são utilizados como entradas de uma rede neural, a qual foi treinada com o objetivo de aumentar a confiança no diagnóstico final. O diagnóstico é mais eficiente e apurado do que o obtido somente usando os filtros de Kalman, entretanto requer um projeto personalizado e o treinamento da rede para defeitos específicos.

Monica Visinsky [Visinsky, 1994] desenvolveu um ambiente de tolerância a falhas dividido em três níveis: um nível básico de controle; um nível intermediário capaz de corrigir falhas sensoriais; e um nível de supervisão capaz de tolerar as falhas nas juntas de manipuladores. O conhecimento para o diagnóstico é armazenado em árvores de falhas em um sistema especialista no nível de supervisão. O sistema é genérico, mas o método de diagnóstico é restrito ao uso de árvores de falhas.

Hamilton et al. [Hamilton et al., 2001] desenvolveram o sistema de diagnóstico RECOVERY, que utiliza redes semânticas particionadas. As redes são utilizadas para integrar informações de diferentes naturezas: informações estruturais equivalentes a árvores de falhas, informações temporais sobre a seqüência de eventos e o registro das falhas observadas. Detectada uma falha, o sistema RECOVERY procura correlações nas informações armazenadas na rede semântica e identifica os possíveis defeitos. É uma ferramenta de uso geral, mas a qualidade do resultado depende diretamente da qualidade da informação contida na rede semântica. O processamento das informações

sobre os eventos e falhas, para inserção na rede, deve ser apurado o suficiente para fornecer dados capazes de diferenciar entre defeitos; em outras palavras, métodos matemáticos são muito necessários se a assinatura das falhas e ou as possíveis causas forem muito próximas.

Métodos de detecção inspirados em sistema biológicos também são utilizados. Um exemplo é o trabalho de Huntsberger [Huntsberger, 1998], no qual toda a informação sensorial é comparada com valores armazenados na memória de curta duração (*STM - Short Term Memory*). Para todos os dados sensoriais são calculados valores de incerteza em relação ao passado recente. Se a incerteza é muito grande, o sensor é considerado defeituoso.

Murphy e Hershberger [Murphy and Hershberger, 1996, Murphy and Hershberger, 1999] desenvolveram um sistema utilizando esquemas perceptivos, chamado de SFX-EH (*Sensor Fusion Effects-Exception Handling*). O sistema utiliza modelos causais parciais dos sensores, do ambiente, e da interação das tarefas, em conjunto com testes ativos utilizados para classificar ou distinguir as falhas. O SFX-EH explora a propriedade dos robôs serem agentes fisicamente situados no ambiente, portanto capazes de interagir e obter informações específicas, que possibilitem a verificação da validade de hipóteses de falhas. O processo de diagnóstico deixa de ser essencialmente passivo para ser ativo no SFX-EH, permitindo a ampliação do espaço de observação de falhas através da execução de rotinas de teste previamente implementadas no controle.

O robô Hannibal de seis pernas, 19 atuadores e 60 sensores desenvolvido por Cynthia Ferrell [Ferrell, 1994] mostrou importantes características de tolerância a falhas. A confiança de cada sensor é refletida por um indicador de "dor"específico (*pain parameter*). Este indicador é fruto de dois processos. O primeiro processo identifica estados predeterminados de um ciclo do sensor em relação ao ciclo de atuação, e permite a identificação dinâmica de parâmetros de percepção do sensor. O segundo é um monitor de consenso dos sensores, que compara os estados detectados por cada um individualmente e procura valores discrepantes. Nos dois casos, o parâmetro de dor correspondente ao sensor discrepante é incrementado, representando a redução da sua confiança. Quando o nível de dor ultrapassa um determinado limite o sensor é considerado defeituoso. Quando falhas nos sensores de uma perna ou nos atuadores impedem seu funcionamento normal, é considerada uma falha grave ou catastrófica.

4.3 A recuperação de falhas

A recuperação das falhas é a execução da ação adequada para que a falha não provoque um funcionamento errado do sistema, e no caso de robôs, o não cumprimento da missão ou tarefa. A essência da recuperação das falhas é a utilização de recursos redundantes, seja de percepção, de atuação, ou de tempo. As ações de recuperação de

falhas em um robô são intimamente ligadas ao projeto estrutural e do controle. Sendo assim, as ações e mecanismos de recuperação são limitados às restrições construtivas e necessitam ser planejadas e inseridas previamente [Ferrell, 1994, Payton et al., 1992]. O controle deve selecionar a ação de recuperação adequada, o que depende do diagnóstico correto. É importante ressaltar que a tolerância a falhas não amplia as capacidades existentes em um sistema, basicamente oferece recursos de reconfiguração adequados à presença de defeitos e falhas, aumentando a sua confiabilidade e disponibilidade. Muitas alternativas utilizadas na tolerância a falhas causam degradação no desempenho do sistema, quando não existem módulos totalmente redundantes disponíveis. No caso de um robô individual, normalmente são utilizadas as seguintes abordagens para implementar a tolerância e recuperação de falhas: processamento redundante no controle; sensores redundantes; atuadores redundantes; manipuladores com redundância cinemática; manipulador redundante; plano ou solução alternativa. Cada uma delas é descrita a seguir.

Processamento redundante: A detecção e recuperação de falhas de processamento no controle é implementada normalmente com o uso de sistemas multiprocessados, programação N-versões, execução repetida do mesmo módulo e uso de sistemas supervisores. A tolerância a falhas no controle pode incluir as falhas de software e de hardware. As falhas de software podem ser toleradas através do uso de múltiplas versões dos processos e de mecanismos de verificação. As falhas de hardware no controle são somente suportadas através do uso de uma plataforma multiprocessada adequada.

Os métodos de detecção e recuperação de falhas no controle são os mesmos utilizados genericamente em sistemas de tempo real. Essencialmente, cópias primárias e secundárias dos processos são distribuídas em vários processadores [Hecht et al., 2000, Bertossi et al., 1999, Shokri and Beltas, 2000, Liberato et al., 2000, Mili et al., 1998]. Quando as restrições de tempo de um processo são críticas, são executadas múltiplas instâncias em diferentes processadores simultaneamente. Para tarefas que podem aceitar um atraso maior na recuperação, são executadas cópias redundantes apenas em condições de presença de falhas.

Uma abordagem interessante na tolerância do controle para robôs é a utilização de um módulo de supervisão assistido por um operador funcionando em conjunto com módulos de controle replicados em hardware [Marzwell et al., 1994]. Os módulos replicados em hardware garantem a tolerância as falhas de hardware, do sistema operacional e das aplicações em tempo real. O módulo de supervisão previne falhas de alto nível como a colisão com objetos. Entretanto, a utilização de módulos de controle totalmente redundantes e de módulos de supervisão associados nem sempre é viável em robôs autônomos.

No trabalho de Lueth e Laengle [Lueth and Laengle, 1994] foi desenvolvido um

sistema de controle totalmente distribuído. Concluíram que sistemas distribuídos facilitam a implementação de redundância nos agentes, entretanto complicam os aspectos de recuperação de falhas que envolvem a centralização de informações, como é o caso do diagnóstico.

Sensores redundantes: A redundância de sensores é muito utilizada em robôs com o intuito de melhorar a precisão da percepção, normalmente sem a preocupação direta com a tolerância a falhas. A essência da recuperação de falhas sensoriais é a seleção e utilização de fontes alternativas da mesma informação, ou seja, em um dado instante, desprezar os dados dos sensores pouco confiáveis e valorizar os dados dos mais confiáveis. Em muitos sistemas robóticos isto já é realizado diretamente no processo de fusão sensorial, como parte normal do controle.

No controle deliberativo, a remoção dos dados dos sensores defeituosos muitas vezes é realizada no próprio processo de construção do modelo de mundo, desprezando dados incoerentes. No modelo comportamental os dados dos sensores reais são isolados através do uso de sensores virtuais [Ferrell, 1994, Payton et al., 1992, Murphy and Hershberger, 1999]. Os sensores virtuais agregam informações de vários sensores reais e são reconfigurados de acordo com o estado corrente de falhas. Os processos de detecção de falhas, recalibragem de parâmetros, reconfiguração e reintegração dos sensores reais são normalmente encapsulados dentro dos sensores virtuais.

As fontes de dados utilizadas nos sensores virtuais podem ser diferentes. Por exemplo, considere dois sensores em uma junta, um de velocidade e o outro de posição, e ainda dois sensores virtuais de velocidade e posição utilizados pelo controlador. No funcionamento sem falhas as informações dos sensores virtuais são obtidas diretamente dos sensores físicos. Caso o sensor real de velocidade falhe, o sensor virtual pode continuar funcionando, derivando a saída do sensor de posição. O mesmo pode ser realizado para o sensor de posição através da integração da velocidade. É importante destacar que a transformação de uma informação em outra pode exigir processamento e modificar o tempo de resposta do sensor virtual.

Atuadores redundantes: A instalação de dois motores em uma mesma junta ou em uma mesma roda são exemplos de atuadores redundantes. Por dificuldades construtivas é uma solução muito pouco utilizada na prática, entretanto é a mais fácil de ser aplicada na recuperação de falhas para o controle. Este simplesmente passa a enviar os comandos para o atuador extra, ao mesmo tempo em que desativa o atuador principal.

Redundância cinemática: O uso de redundância cinemática em manipuladores já é amplamente estudado. Cada junta de um manipulador corresponde a um grau de liberdade (DOF, *degree of freeedom*). Um manipulador é dito redundante quando

é capaz de alcançar o mesmo ponto no espaço com configurações diferentes. Quando os motores ou os sensores de uma junta apresentam defeitos, a mesma é usualmente travada. Com a junta fixa em determinada posição, o controlador do robô continua a definir os movimentos com um DOF a menos. A continuidade da tarefa nem sempre é possível, pois depende da capacidade do robô em alcançar todas as posições necessárias em função da posição das juntas defeituosas. Existem muitos trabalhos que visam melhorar a tolerância cinemática usando o mínimo necessário de recursos extras.

Paredis e Khosla [Paredis and Khosla, 1994] apresentam um método matemático para definir o espaço de trabalho tolerante a falhas de manipuladores N-DOF na presença de K falhas. Maciejewski e Lewis [Lewis and Maciejewski, 1994] definem medidas de tolerância de um manipulador utilizando análises para piores posições possíveis para as juntas. Paredis e Khosla [Paredis and Khosla, 1996] propõem um algoritmo para controle de trajetórias na execução de tarefas; este minimiza a probabilidade do travamento de juntas nas piores posições, maximizando assim, a tolerância a falhas cinemática do manipulador. Liu [Liu, 2001] propõe um método integrado que une a estimação de parâmetros, as leis de controle, a tolerância a falhas do atuador e a detecção de falhas em um mesmo algoritmo.

Manipulador redundante: A recuperação de falhas com manipuladores redundantes é encontrada em alguns robôs, na forma de rodas, pernas ou braços mecânicos extras. A recuperação de falhas, neste caso, envolve a transferência da tarefa corrente de um manipulador para outro pelo controlador. Este seleciona um novo plano ou ação por parte do controlador que utiliza o manipulador redundante.

Plano ou solução alternativa: Mesmo quando não existe redundância específica ou equivalente nos manipuladores e atuadores de um robô, este ainda pode ser capaz de realizar a tarefa desejada utilizando métodos ou abordagens diferentes. Com a existência de planos alternativos, em alguns casos pode ser possível aumentar a disponibilidade de um sistema robótico na execução de uma tarefa, mesmo apresentando degradação de desempenho. Pode-se dizer que o uso de manipuladores redundantes e de redundância cinemática são especializações na criação de planos alternativos.

Em um controle deliberativo um novo plano de ação pode ser escolhido, de forma a não utilizar os elementos defeituosos. No controle comportamental, os comportamentos prejudicados pelas falhas devem ser inibidos e outros alternativos podem ser ativados. A essência da questão de adaptação do controlador às falhas é realizar o melhor esforço no objetivo de continuar a realizar a tarefa ou missão sem utilizar os elementos defeituosos, ou utilizando os recursos disponíveis. Muitas vezes, na presença de um defeito ou perda da confiabilidade em um determinado módulo, o controle deve alterar parâmetros internos ou o modo de realizar uma determinada ação para

garantir uma situação mais segura para o sistema. Pode-se, por exemplo, adotar posturas mais cuidadosas reduzindo velocidade ou força de atuadores [Ferrell, 1994].

4.4 Arquitetura de controle híbrida

Este trabalho se iniciou com foco em abordagens baseadas em comportamentos, mas ao longo do tempo este foco foi redirecionado para arquiteturas híbridas. Neste processo de pesquisa, muitos dos requisitos pertinentes ao modelo desenvolvido tiveram sua origem em trabalhos de abordagens totalmente ou parcialmente baseadas em comportamentos. Nesta seção encontram-se os trabalhos que forneceram subsídios e soluções que inspiraram esta tese.

As abordagens de controle baseado em comportamentos apresentam vantagens e desvantagens para a implementação da tolerância a falhas. A modularidade e simplicidade dos comportamentos e da comunicação entre eles facilitam o processamento distribuído e consequentemente, o uso de arquiteturas multiprocessadas. Esta característica é adequada para implementação de métodos de tolerância para falhas de software e de processamento. Outro fator favorável é a interação direta dos comportamentos com os dados sensoriais que facilita a utilização de redundância de sensores, sem envolver métodos elaborados de modelagem do mundo.

A inteligência apresentada pelos sistemas de controle comportamental emerge da interação existente entre os comportamentos. Esta interação não é de fácil modelagem ou previsão, o que dificulta a inclusão no controle de planos alternativos que, na presença de falhas, possam alterar significativamente a forma que a tarefa atual está sendo realizada. Este é um dos motivos pelos quais não é comum encontrar na literatura sistemas complexos tolerantes a falhas com abordagem puramente comportamental. A maior parte os trabalhos apresenta características híbridas ou um sistema de supervisão capaz de ativar e desativar comportamentos.

Na arquitetura híbrida, o controle é dividido em dois níveis: o nível com os comportamentos, esquemas ou outras funções que reagem as entradas sensoriais; e o nível deliberativo que normalmente opera ativando e desativando as funções do nível mais baixo de acordo com objetivos ou informações mais complexas. Tanto na abordagem puramente comportamental quanto na híbrida, o nível de controle mais baixo é responsável pela interação mais direta com os sensores e atuadores, sendo submetido às restrições mais fortes de tempo.

A especialização no uso dos dados sensoriais por cada comportamento facilita a implementação dos sensores virtuais. Os sensores virtuais são sensores abstratos que isolam os comportamentos dos sensores reais, oferecendo fontes de dados sensoriais que embutem a tolerância a falhas de sensores de forma transparente. Como os dados utilizados pelos comportamentos são mais simples, o processamento necessário para implementar a tolerância em um sensor virtual também fica mais simples. Caso o

sensor virtual não esteja disponível devido a falhas nos sensores dos quais dependem, apenas os comportamentos dependentes daquela informação vão ser inibidos, ou não vão ser ativados.

Os comportamentos também podem ser especializados em relação aos atuadores. Se um atuador específico apresenta uma falha, os comportamentos que enviam comandos a ele podem ser inibidos. Além disso, é possível criar comportamentos que se ativam em função da detecção ou diagnóstico de uma falha. Estes comportamentos de recuperação podem executar ações corretivas ou prevenir danos sistêmicos ou ambientais, por exemplo, enviar um comando para ativar um freio em uma junta quando o motor conectado não estiver funcionando adequadamente.

Concluindo, as características de modularidade, independência e simplicidade dos comportamentos são fatores que facilitam alguns aspectos da implementação da tolerância a falhas, isto se existir um nível supervisor integrado ao controle do robô capaz de ativar e desativar comportamentos ou funções de controle de baixo nível de acordo com o estado de falhas do sistema.

Estas características apresentadas não são exclusividade de sistemas híbridos com comportamentos. Qualquer abordagem híbrida que utilize no baixo nível um conjunto de funções de controle simples, modulares e com alto grau de independência oferece as mesmas vantagens para a implementação da tolerância a falhas. Alguns dos objetivos que devem ser buscados são os seguintes:

- Responder rapidamente à presença das falhas, evitando danos internos ou externos.
- Permitir uma degradação do desempenho gradual à medida que as falhas se acumulam.
- Utilizar na melhor maneira possível todos os recursos disponíveis.
- Suportar uma grande variedade de falhas, previstas ou não.

4.5 Trabalhos relacionados

Vários trabalhos encontrados na literatura contribuirão para o desenvolvimento desta tese. Muitos já foram citados ao longo deste capítulo, com ênfase em aspectos específicos dos problemas de detecção de falhas, diagnóstico e recuperação de falhas. Nesta seção, são destacados os aspectos mais importantes de cada trabalho que inspiraram características da tese desenvolvida.

Um dos trabalhos mais relevantes de controle híbrido com tolerância a falhas foi o robô submarino não tripulado desenvolvido por Payton et al. [Payton et al., 1992]. Um aspecto chave neste trabalho é que a reconfiguração do robô para tolerar as falhas

é controlada pela avaliação dos resultados percebidos das ações efetuadas pelos comportamentos e não pela identificação direta do defeito. Esta abordagem proporciona ao sistema uma alta capacidade de tolerar falhas, mesmo imprevistas.

O controlador do robô é programado com diversas estratégias diferentes e redundantes para realizar uma mesma tarefa, sendo associados a cada uma um modelo próprio de desempenho. Uma falha é detectada quando o desempenho do comportamento utilizado não é o esperado. Quando uma estratégia não funciona, o controlador seleciona outras sucessivamente até que obtenha um desempenho aceitável. Os comportamentos são divididos em grupos de estratégias associadas aos objetivos da missão. Quando os objetivos se alteram, o conjunto de comportamentos ativos também se altera. A recuperação de falhas é realizada através da seleção dos conjuntos de comportamentos que estão funcionando adequadamente. Isto é realizado através do retorno proveniente do modelo de desempenho, sendo este associado a um valor de participação específico para cada comportamento.

O diagnóstico real da falha não é realizado no trabalho de Payton et al. O controlador identifica apenas os comportamentos que funcionam através de seqüências de tentativas inteligentes. Por isso, esta abordagem não especifica a causa real da falha, operando somente de forma indireta através dos sintomas.

O não conhecimento do real defeito pode provocar um atraso muito grande no processo de recuperação por dois motivos principais: várias estratégias diferentes podem ser afetadas pelos mesmos defeitos e a utilização de cada uma delas no processo de tentativas atrasa a seleção de uma estratégia adequada que funcione efetivamente; a percepção da falha é realizada pela observação do desempenho, o que envolve normalmente a percepção e analise da interação do robô com o ambiente. O tempo entre o início da falha e sua detecção de forma indireta pode ser muito grande.

O controle implementado no robô Hannibal por Ferrell [Ferrell, 1994] implementa a tolerância de uma forma diferente. As falhas são classificadas em locais e catastróficas. As locais correspondem a falhas nos sensores e são isoladas através dos sensores virtuais, não alterando a estratégia de controle. As catastróficas correspondem à falhas em algum dos sensores virtuais ou na atuação de uma das pernas. Uma falha catastrófica altera imediatamente os parâmetros do comportamento específico para caminhar, assim mudando significativamente a forma que este controla as pernas. Este comportamento implementa vários passos diferentes capazes de tolerar falhas de uma ou mais pernas. O nível de controle mais baixo detecta as falhas catastróficas, e alerta o nível superior. Este reage imediatamente, alterando os parâmetros de controle do comportamento de caminhar. Ou seja, foi desenvolvido um comportamento adaptável à presença de falhas catastróficas nas pernas.

Os sensores virtuais do Hannibal também são capazes de calibrar os sensores reais e a detecção de falhas destes, automaticamente. Esta calibração é realizada com o auxílio de um modelo cíclico e discreto de eventos perceptivos esperados. A

freqüência de execução dos comportamentos também é adaptável em função da taxa de amostragem dos sensores virtuais.

O sistema SFX-EH de Murphy e Hershberger [Murphy and Hershberger, 1996, Murphy and Hershberger, 1999] é uma arquitetura híbrida de controle reativo e deliberativo e foi chamado de SFX(*Sensor Fusion Effects architecture*). O nível reativo é responsável pela execução do plano, controlando todas as atividades que necessitam de informação referentes às tarefas do robô. É dividido em duas partes: um gerente de tarefas que ativa os comportamentos; um gerente de sensoriamento que aloca e controla os recursos perceptivos. Cada comportamento é dividido em um esquema perceptivo e um esquema motor. O esquema motor produz as saídas correspondentes às ações do comportamento, enquanto o esquema perceptivo é responsável por fornecer informações confiáveis selecionadas a partir de um conjunto de sensores lógicos. Os sensores lógicos ou virtuais correspondem a sensores reais acoplados através de algum algoritmo de processamento. A padronização dos dados provenientes dos sensores lógicos possibilita a reconfiguração automática dos esquemas perceptivos, ou seja, os esquemas perceptivos são capazes de receber informações de alguns sensores lógicos diferentes.

Quando uma falha é detectada no nível reativo, o esquema perceptivo envia uma exceção ao gerente de sensoriamento. Identificada a falha, duas ações podem ser executadas. O gerente de tarefas pode ser informado da falha para que possa modificar o plano corrente e conseqüentemente os comportamentos ativos. Ou o esquema perceptivo pode receber a identificação de um sensor lógico alternativo. Caso a identificação da falha não seja possível, uma lista de hipóteses é gerada e uma seqüência de testes é iniciada com o intuito de apurar o diagnóstico. Associada as hipóteses estão as ações de recuperação. Se o sistema identifica unicamente o processo de recuperação, este é executado mesmo que não se tenha chegado a um diagnóstico preciso. Se não for possível a recuperação ou o uso de comportamentos alternativos a lista de tarefas do nível deliberativo é alterada.

O projeto de Huntsberger [Huntsberger, 1998] foi destinado a robôs com intuito de explorar outros planetas, assim sendo, foi considerada a possibilidade de redundância muito restrita. Uma estrutura de controle hierárquica acoplada a uma análise adaptativa dos sensores foi adotada. A tolerância é implementada na percepção com a alteração de parâmetros de contribuição dos sensores na tomada de decisões de cada comportamento em função do grau de incerteza. Praticamente não existe tolerância a falhas de atuadores e manipuladores neste projeto; o enfoque principal foi prevenir e evitar situações de risco, tais como ficar preso em uma região com sombra ou muito irregular durante a noite.

Os métodos de detecção de falhas diretos como os implementados no Hannibal e SFX-HE e indiretos como no trabalho de Payton não são exclusivos, mas sim, complementares. Os métodos diretos proporcionam eficiência e rapidez na detecção

e recuperação de falhas, enquanto os métodos indiretos proporcionam uma melhor adequação às condições ambientais, além de ampliar o espaço de observação do diagnóstico, permitindo assim, uma melhor reação a falhas não previstas.

Um fator muito importante é a determinação das capacidades disponíveis no robô em função do estado de falhas. No sistema SFX-EH existe uma relação entre as capacidades do robô de realizar tarefas específicas e o conjunto de comportamentos disponíveis e confiáveis. Desta relação, pode ser possível se associar às tarefas indicadores de desempenho e confiabilidade, que permitam uma melhor seleção das atividades e uso dos recursos disponíveis. Vale ressaltar que para definição de tarefas não é necessário um modelo deliberativo ou híbrido [Maes, 1990].

4.6 Times de robôs cooperativos

Atualmente na comunidade de robótica existem muitos trabalhos sendo realizados na área de times cooperativos. O uso de um time para realizar determinadas missões se mostra uma abordagem mais robusta, escalonável e muitas vezes mais econômica em relação ao desempenho obtido, com apenas um robô.

O uso de times também amplia as possibilidades de implementação de tolerância a falhas de um sistema robótico. A redundância necessária para tolerar determinadas falhas pode não estar presente na competência individual de um robô, mas distribuída entre os integrantes do time ou na cooperação entre eles.

Quando se estuda a tolerância a falhas de um robô individualmente, o enfoque é garantir que o mesmo complete a missão usando todos os recursos possíveis, ou seja, nunca desistir. No caso de um time, este problema é um pouco diferente, pois é possível que haja troca de papéis ou substituições ao longo do tempo. O objetivo individual é fazer o melhor para o grupo e não necessariamente o melhor em uma tarefa específica.

Um robô pode possuir um conjunto de competências individuais correspondendo às tarefas que é capaz de executar. Dois robôs podem ter a mesma competência, mesmo que seja implementada de forma completamente diferente. Abordagens diferentes para realização de uma mesma tarefa podem significar a possibilidade do time melhor se adequar a ambientes diferentes ou a variações do mesmo ao longo do tempo.

Se um robô sofre uma falha e não é mais capaz de executar uma determinada tarefa, a mesma deve ser transferida para outro elemento do time mais apto para realizá-la. Caso nenhum outro robô seja capaz de assumir a tarefa, pode ser ainda possível realizá-la, através da estreita cooperação entre dois ou mais integrantes do time.

A tolerância a falhas vista a este nível é equivalente aos problemas de adaptabilidade de missões em times cooperativos: como distribuir e redistribuir da melhor forma possível as tarefas ou sub-tarefas entre os integrantes do time, seja utilizando

cooperação estreita ou não. Um sistema de controle de um time capaz de redistribuir tarefas quando um robô é incluído ou retirado já apresenta tolerância a falhas, quando se considera a perda completa de um robô como parte do modelo de falhas. Entretanto, para que times cooperativos apresentem um modelo de falhas mais robusto e genérico é necessário considerar outros importantes aspectos: a comunicação entre os integrantes pode falhar, a percepção de falhas individual ou coletiva pode falhar, e o desempenho dos integrantes ou do time pode se alterar em função do tempo ou do ambiente.

As tarefas de uma missão são repartidas para o time em função das competências e desempenhos individuais. Como estas se alteram no tempo em função de falhas e de variações no ambiente, a realocação de tarefas deve ser realizada dinamicamente, para garantir a otimização do desempenho do time e muitas vezes até o cumprimento da missão. Além disso, para maximizar a chance de sucesso na missão tolerando também falhas de comunicação, temporárias ou não, os integrantes do time devem possuir seqüência de tarefas a serem realizadas ou capacidade individual para decidi-las.

Para que um time possua robustez na distribuição de tarefas, é necessário que tolere a perda de qualquer membro, ou da comunicação entre eles. Para tanto, a alocação de tarefas deve funcionar de forma transferida entre os membros ou funcionar de forma distribuída. Permitindo que cada robô tome decisões individuais e isoladas se necessário.

O desempenho na execução de uma tarefa é um parâmetro muito difícil de ser avaliado, principalmente na presença de alguma falha que pode não ser percebida. Caso um robô não esteja realizando a sua função adequadamente, pode ser necessário que um companheiro do grupo o informe do fato ou o substitua. Em prol da missão comum o robô deve aceitar o processo de substituição.

Um trabalho muito significativo sobre tolerância a falhas em times de robôs é o ALLIANCE desenvolvido por Lynne E. Parker [Parker, 1994, Parker, 1997, Parker, 1999]. O ALLIANCE funciona de forma distribuída em times heterogêneos, onde cada robô realiza uma seleção adaptativa de suas tarefas em função de motivações modeladas matematicamente, como a impaciência e a aquiescência (*impatience and acquiescence*).

Cada robô é capaz de realizar um conjunto de tarefas predeterminadas e conhece o desempenho esperado na execução normal para cada uma. Se um robô observa um outro realizando uma tarefa de que é capaz e percebe que o outro está demorando muito mais que o esperado, vai aumentando o seu nível de impaciência. Quando o nível de impaciência ultrapassa um determinado limite, o robô força a troca de tarefas. O robô substituído consente, ajusta um parâmetro de aquiescência, e tenta realizar uma nova tarefa. O parâmetro de aquiescência é utilizado para o robô não tentar realizar a tarefa novamente.

O ALLIANCE atende muitos dos requisitos necessários a tolerância a falhas em times de robôs. O mecanismo de motivação funciona de forma distribuída tolerando algumas falhas de comunicação. Quando um robô desiste ou é substituído por outro em uma determinada tarefa, ele inicia uma outra com o intuito de manter sua utilidade. Como a causa da perda de desempenho não é analisada, o robô pode tentar executar tarefas para as quais também não está mais apto. Esta atitude pode gerar um atraso na execução destas tarefas por outros robôs. O conhecimento da degradação do desempenho em função da falha detectada é importante em qualquer processo de realocação. A redefinição de uma tarefa como a mudança do tamanho de um território também não é um processo simples no ALLIANCE, já que pode influir diretamente no desempenho da tarefa.

Na continuação do trabalho, Parker incluiu o processo de treinamento já desenvolvido [Parker, 1997] durante a execução da missão, criando assim L-ALLIANCE [Parker, 2000a]. Os parâmetros de desempenho se ajustam ao longo do tempo, juntamente às prioridades das tarefas em função de alterações da missão.

Em 1998 Schneider-Fontan e Mataric [Schneider-Fontan and Mataric, 1998] demonstraram a abordagem de controle baseado em comportamento para um grupo de robôs móveis. A tarefa foi associada a um território capaz de ser alterado dinamicamente. O sistema se adapta a perda e a inclusão de elementos no time simplesmente alterando os territórios assinalados a cada um. A divisão em territórios minimiza a interferência entre os robôs e permite que cada um faça a própria alocação da tarefa independentemente e de forma determinística, conhecendo apenas quais robôs falharam. O método de tolerância a falhas é eficiente, mas é associado a uma tarefa com propriedades territoriais.

No controle de um conjunto de robôs para "Robocup soccer" Werger [Werger, 1999] colocou a tolerância como um objetivo de projeto dos comportamentos do controle. O trabalho é minimalista no sentido de ativar os comportamentos com o mínimo de informação necessária e de reter o mínimo de tempo possível os dados sensoriais. Os comportamentos foram desenvolvidos com o intuito de funcionar com altos níveis de incertezas. A tolerância a falhas nos sensores foi implementada aceitando as incertezas e não tentando eliminá-las. O sistema se mostrou robusto em relação às incertezas, mas a abordagem minimalista não é fácil de ser implementada em sistemas complexos.

Uma abordagem de cooperação estreita com tolerância a falhas é apresentada por Gerkey e Mataric [Gerkey and Mataric, 2001] na tarefa de se carregar uma caixa em conjunto. Cada robô possui um conjunto de competências utilizadas para realizar as tarefas. As tarefas são alocadas através de um processo de leilão distribuído, no qual o robô que se acha mais apto para a tarefa ganha. Não existe um elemento centralizador do leilão, o que poderia comprometer a robustez do time inteiro. A alocação de tarefas é pelo sistema MURDOCH [Gerkey and Mataric, 2000]. Quando

uma tarefa é introduzida por um operador ou pelo próprio time, é iniciado um leilão para sua alocação. A tarefa é publicada para o time na forma de conjunto de assuntos de um espaço predefinido e conhecido por todos (*Subject Namespace*). Estes assuntos determinam recursos necessários, estados e atributos da tarefa a ser realizada. Cada robô avalia o pedido, calcula um índice de desempenho e responde publicamente o seu lance. O robô que fez o melhor lance assume a tarefa.

O MURDOCH facilita a gerência global de tarefas permitindo a inclusão e remoção de membros com competências diferentes em qualquer momento da missão. Esta característica o torna mais flexível que o ALLIANCE para reconfiguração do time e das tarefas disponíveis. Entretanto, é mais suscetível a falhas de comunicação.

Uma abordagem para alocação de tarefas baseada em árvores binárias em um time com tolerância a falhas foi proposta por Haihang Sun e Robert McCartney [Sun and McCartney, 2001]. Toda a informação sobre a divisão das tarefas e alocação das mesmas é armazenada em uma árvore binária. Esta árvore é compartilhada por todos e permite aos integrantes do time realizarem decisões de balanceamento de carga de forma rápida e independente, sem utilizar processos de negociação. As tarefas são decompostas hierarquicamente em sub-tarefas e os robôs são associados às folhas e indiretamente as sub-árvores. Toda vez que um robô troca sua posição ou acaba uma tarefa informa a todos os outros, para que possam atualizar a informação na árvore. Quando algum integrante do time não envia mensagens de vida ("*I am alive*") é considerado em falha e a árvore dos outros membros é atualizada para representar a nova composição.

Quando um robô percebe grandes diferenças de balanceamento de carga entre os ramos da árvore, ele pode decidir trocar de tarefa. Escolhe primeiramente entre as tarefas irmãs, depois primas e por último se necessário assume uma tarefa pai que agrupa todos as tarefas filhas. O trabalho mostra uma maneira de distribuir a carga em um time aceitando falhas completas dos elementos, minimizando a necessidade de comunicação. A decomposição de tarefas hierarquicamente pode se vista como uma extensão do conceito territorial apresentado com apresentado por Fontan [Schneider-Fontan and Mataric, 1998].

4.7 Considerações gerais

O diagnóstico de falhas nos robôs é um problema muito complexo devido ao reduzido espaço de observação, quando comparado à infinidade de possibilidades de estados e interações diferentes entre o robô e o ambiente. Os métodos genéricos como árvores de falhas são aplicáveis, mas exigem complementos através de métodos numéricos específicos e análise de conhecimentos variados para se obter um diagnóstico satisfatório, principalmente se for necessário distinguir as falhas mecânicas e sensoriais. O treinamento específico dos métodos de detecção ou o uso de modelos funcionais

muito refinados que incluem além da descrição do sistema, a sua interação com o meio, também podem ser necessários para a implementação satisfatória da tolerância a falhas. Além disso, o profundo conhecimento do projeto se mostra essencial na construção dos modelos e métodos de detecção e diagnóstico de falhas.

O modelo de controle baseado em comportamentos coloca os sensores próximos dos comportamentos que definem as ações. As falhas nos sensores podem ser isoladas através do uso de uma abstração em relação aos sensores reais, chamados sensores virtuais. Estes sensores virtuais ou lógicos oferecem aos comportamentos dados confiáveis através da seleção do método de processamento e dos sensores reais mais adequados ao momento.

A redundância de sensores e dos métodos de processamentos para converter ou fundir os dados sensoriais necessita do conhecimento do projetista do sistema, mas a reconfiguração dinâmica dos sensores em função dos diagnósticos dos sensores reais pode ser realizada por mecanismos genéricos.

A recuperação de falhas nos atuadores ou manipuladores necessita que o controle realize decisões alternativas, quando detectadas falhas que impossibilitem as ações normais. A implementação de tolerância a falhas de atuadores em controles baseados em comportamento é realizada com inibição e ativação de comportamentos alternativos para os mesmos objetivos. O problema de definir comportamentos alternativos é equivalente à composição do próprio controle do robô, já que os comportamentos definem as interações entre os estímulos e as ações compatíveis. A definição das possíveis alternativas precisa ser realizada com o conhecimento do projetista do controle, que também deve especificar as incompatibilidades existentes.

A descrição de um controle pode conter grande parte das dependências existentes entre os comportamentos e o hardware utilizado para atuação ou percepção. As dependências podem definir em função do estado de falha do sistema os comportamentos que estão operacionais ou não.

Na maioria dos sistemas de tempo real, as restrições de tempo na amostragem e no processamento de informações sensoriais são definidas por características específicas do aspecto físico monitorado. Esta relação entre os comandos enviados e a taxa de amostragem de sensores pode ser utilizada como um grau de liberdade extra na definição das restrições temporais de alguns processos.

Se for impossível controlar um atuador na velocidade normal porque o controlador não consegue garantir as restrições de tempo no processamento do controle, por que não diminuir a velocidade do atuador. Em outras palavras, se o controlador está gastando mais tempo para "pensar" devido a falhas, porque não andar mais devagar. Esta abordagem pode permitir ao sistema manter a confiabilidade mesmo em situações muito adversas. Este é um problema complicado que envolve a adaptação dinâmica de restrições de tempo e dos comandos enviados aos atuadores. Estas relações normalmente são definidas pelo projetista, mas a implementação pode

ser facilitada pela integração dos comportamentos e o controle de processamento.

Os métodos de recuperação de falhas do hardware e o diagnóstico dependem do conhecimento específico do projetista sendo sempre específicos. Apesar disto, se os comportamentos e os processos perceptivos utilizarem interfaces de comunicação padronizadas, é possível utilizar mecanismos genéricos para seleção e ativação dos métodos de recuperação no controle.

Capítulo 5

Modelo Proposto

> Não existe nenhum caminho lógico para a descoberta das leis elementares do Universo; o único caminho é a intuição.
>
> Albert Einstein (1879 - 1955)

As pesquisas sobre as abordagens de controle dos robôs permitiram avanços muito concretos na ultima década, entretanto as discussões e novas alternativas ainda estão longe de terminar. Pode-se dizer que é um campo em constante evolução. Dentro deste quadro, o trabalho realizado nesta tese visou criar uma metodologia de desenvolvimento do controle de um robô que facilite a inclusão da tolerância a falhas adaptativa sem restringir em demasiado as abordagens disponíveis para o projetista.

É proposto um modelo de arquitetura híbrido, no qual o nível mais alto do controle é definido por uma máquina de estados finitos. Para o nível mais baixo é definida a estrutura de programação na forma de um fluxo de dados. As interações entre os dois níveis e entre os processos de tempo real são também especificadas dentro do modelo. Entretanto, a abordagem para o controle de mais baixo nível fica aberta para decisão ou preferência do projetista.

A utilização do modelo em um ambiente completo de programação deve proporcionar um ciclo menor de desenvolvimento e teste de sistemas com tolerância a falhas adaptativa. A máquina de estados finitos do alto nível do controle facilita a inclusão de novos estados redundantes e estados de teste, facilitando o desenvolvimento e a realização de testes de forma incremental. A regularidade e modularidade do nível mais baixo do controle na forma de um fluxo de processamento facilitam a reutilização de código, inclusive das funções de detecção de falhas e realização de diagnósticos. O *overhead* provocado pelo uso da estrutura de controle do nível mais baixo é o ponto negativo do modelo. Estes pontos serão detalhados no decorrer do texto.

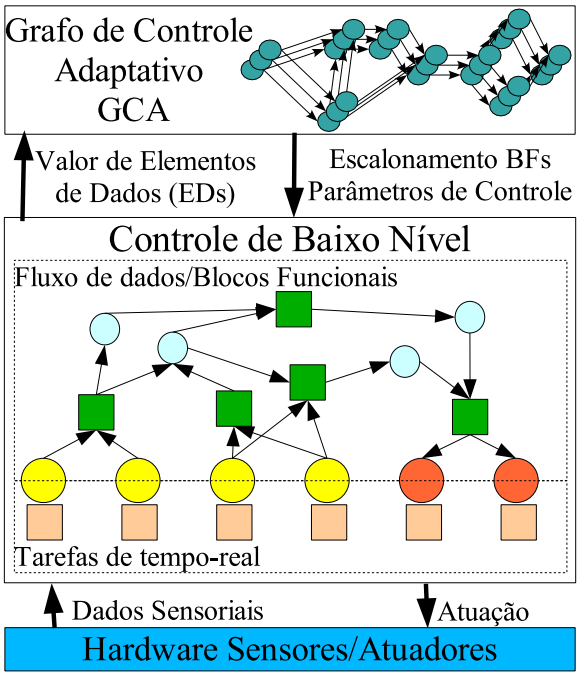


Figura 5.1: Estrutura hierárquica de controle utilizada.

A primeira etapa no desenvolvimento de um modelo e metodologia para tolerância a falhas adaptativa foi definir, de forma mais genérica possível, os seus requisitos. O estudo da literatura existente permitiu a definição das características essenciais ou desejáveis para os controles de robôs tolerantes a falhas. Estas características ou requisitos foram explicitados junto à descrição mais detalhada do modelo, para facilitar o entendimento de sua aplicabilidade.

O objetivo principal de utilização modelo é oferecer uma estrutura de programação na qual seja fácil integrar recursos básicos e conhecidos de tolerância a falhas com soluções específicas ou personalizadas. Para tanto, deve ser possível utilizar os vários tipos de redundância existentes de maneira simples e padronizada. Estas características favorecem a redução do custo tanto de desenvolvimento como de execução.

O modelo é dividido em dois níveis e da mesma forma é dividida a inserção das redundâncias necessárias e dos recursos de tolerância a falhas. No nível mais alto, implementado por uma máquina de estado finita são incluídos estados de recuperação e estados seguros, além de planos alternativos para a execução de uma tarefa. No nível mais baixo implementado pelo fluxo de dados são tratadas as redundâncias nas informações perceptivas existentes. A estrutura híbrida do modelo desenvolvido é mostrada na Figura 5.1.

5.1 Visão Geral

As arquiteturas com abordagens híbridas buscam unir a simplicidade e eficiência de máquinas de estados finitas, capazes de definir seqüências e tarefas complexas, com processos de processamento perceptivos e de ação eficientes e muitas vezes específicos.

O nível mais alto do controle é definido por uma ou mais missões distintas. A cada missão são associadas seqüências de fases (Seção 7.1). O robô é capaz, portanto de realizar um conjunto de missões, sendo cada uma definida por um autômato composto por fases específicas. As transições entre as fases são definidas por condições realizadas sobre os valores gerados pelo fluxo de processamento no nível mais baixo do controle. A seleção da missão corrente é proveniente de um comando externo ao modelo, permitindo a sua inclusão em times de robôs com mecanismos próprios de distribuição de tarefas.

O nível mais baixo do controle é implementado através de um fluxo de dados ou fluxo de processamento. Esta abordagem não é nova para sistemas de controle [Laplante, 1997], mas encontramos algumas vantagens em sua utilização para implementação de sistema com tolerância a falhas adaptativa. Estas vantagens são descritas em detalhes nas Seções 6.1 e 6.3. O processamento é estruturado utilizando vários blocos de programas ou funções interconectadas. Estas funções foram chamadas no modelo de Blocos Funcionais *BFs* e se comunicam utilizando um conjunto de identificadores abstratos chamados de Elementos de Dados *EDs*. A estrutura básica é mostrada na Figura 5.2. Os *BFs* são interconectados de forma indireta através dos *EDs* de forma a constituir um fluxo de dados. Uma execução do fluxo de processamento corresponde a um ciclo de percepção, decisão e ação. No modelo proposto o projetista define os *EDs* de entrada e de saída de cada *BF*. Esta informação define uma topologia de interconexão correspondendo a um grande fluxo processamento de dados.

Os elementos de dados *EDs* são responsáveis pela interface ou conexão dos diversos blocos funcionais, e podem estar associados a elementos de hardware como sensores ou atuadores, ou a qualquer informação necessária no fluxo, como é visto na Figura 5.3. O acesso aos *EDs* de dentro dos *BFs* é realizado através de uma biblioteca de funções ou *API* desenvolvida com este fim. As funções da *API* utilizam para a identificação de cada *ED* do sistema uma constante inteira, no intuito de tornar a interface simples e o mais eficiente possível durante a execução do controle.

Os *BFs* podem ainda acessar um conjunto de Parâmetros de Controle (*PCs*), os quais serão mais bem explicados na Seção 6.4. Estes parâmetros podem ser utilizados como uma interface de comunicação entre os próprios *BFs* e o controle adaptativo, além de facilitar ajustes internos aos *BFs* em função de alterações internas ou externas ao sistema.

Uma plataforma de execução chamada de Plataforma de Controle Adaptativo

(*PCA*) foi desenvolvida para controlar o escalonamento dos *BFs*, para realizar as funções de comunicação e sincronismo dos *EDs* e *PCs*, e para selecionar as configurações otimizadas pelo controle adaptativo.

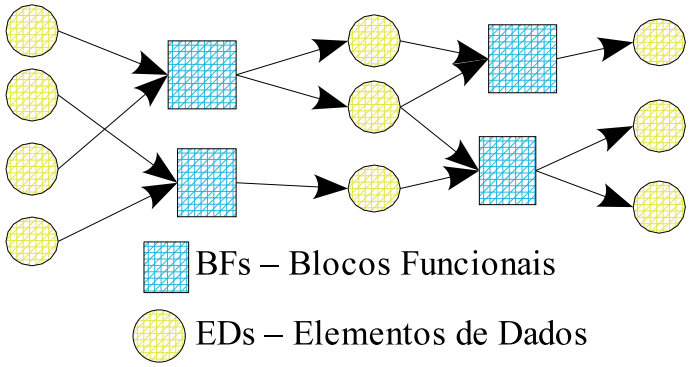


Figura 5.2: Fluxo de dados definido pela interconexão entre os Blocos Funcionais *BFs* e Elementos de Dados *EDs*

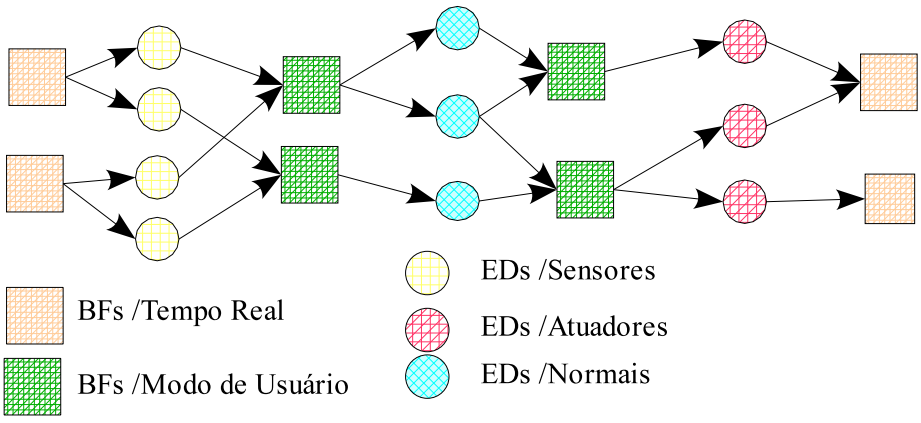


Figura 5.3: Mapeamento dos *EDs* em elementos de hardware existentes, como atuadores e sensores

A integração entre o fluxo de processamento e a máquina de estado do nível mais alto é realizada de maneira simples. A cada fase de uma missão é associado um conjunto de *BFs* que realizam o controle de baixo nível desejado. Os *BFs* selecionados devem produzir valores para os *EDs* associados aos atuadores. Em outras palavras,

a cada fase de uma missão são associadas as funções de código que vão gerar os comandos para a atuação do robô. Por analogia a outros sistemas híbridos, podemos dizer que cada etapa ativa um conjunto de comportamentos, ou um conjunto de esquemas motores, ou um conjunto de equações dinâmicas.

Os *EDs*, correspondentes às entradas dos *BFs* selecionados em uma fase, podem não estar associados diretamente aos sensores do robô. Neste caso pode ser necessário selecionar sucessivamente outros *BFs* que irão completar o fluxo, criando assim, um caminho entre os sensores e os atuadores.

A redundância necessária para a tolerância a falhas nos sensores é inserida no baixo nível do controle, através da composição dos *BFs* e *EDs* e de seus dos atributos. Como já foi citado, o projetista define os *EDs* de entrada e saída de cada *BFs* definindo uma topologia de interconexão. Caso existam múltiplos caminhos para se obter a mesma informação, significa a existência de redundância no fluxo, e neste caso, dois ou mais *BFs* produzem no mesmo ciclo valores para o mesmo *ED*. Esta redundância pode ser utilizada tanto para detecção, quanto para recuperação de falhas no *ED*.

A liberdade no escalonamento dos *BFs* para compor a percepção e processamento de uma fase juntamente com a presença de caminhos redundantes pode ser utilizada para criar as configurações necessárias para a implementação da tolerância a falhas nos sensores ou no processamento das informações internas. Se os caminhos alternativos para gerar a mesma informação associada a um *ED* utilizarem recursos diferentes, o sistema pode ser capaz de tolerar falhas nos recursos disjuntos, sejam de software ou de hardware.

Para simplificar a execução do fluxo, cada composição de *BFs* associada a uma fase específica é armazenada na forma de uma seqüência ou escalonamento de identificadores de *BFs*. Este escalonamento implementa na prática a topologia desejada, pois pode ser gerado respeitando todas as dependências de dados existentes no fluxo.

Na Figura 5.4 é exemplificado este conceito de redundância no fluxo. A topologia completa dada pelo projetista é definida no item (1), sendo que o ED_5 é gerado por dois *BFs* diferentes ($F1$ e $F2$). Como a informação necessária possui duas fontes distintas, é possível criar um fluxo completo ativando apenas um *BF* de cada vez, obtendo assim, mais duas configurações (2 e 3). Cada uma das configurações possíveis pode apresentar custos de processamento e confiabilidade diferentes, além de tolerar conjuntos de falhas diferentes associadas aos *EDs* de entrada ou aos *BFs* utilizados. Este exemplo bem simples já apresenta a possibilidade de fluxo com três configurações diferentes $(1, 2, 3)$.

O modelo concentra a redundância nos *EDs* e conseqüentemente concentra também as funções básicas de detecção e recuperação de falhas. São associadas aos *EDs* funções de teste e comparação de dados. Essencialmente, estas funções detectam valores incoerentes e selecionam os valores mais prováveis de estarem corretos, de acordo com critérios preestabelecidos. Os resultados dos testes realizados nos *EDs*

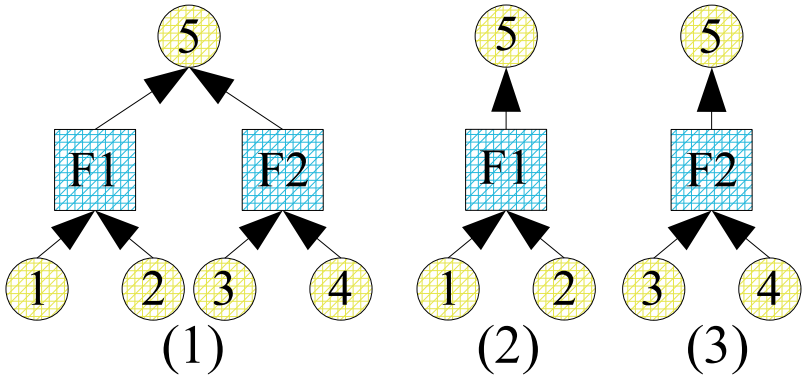


Figura 5.4: Exemplo de topologias redundantes.

podem ser analisados por métodos ou sistemas de diagnóstico.

Toda a estrutura básica do controle de baixo nível implementado pelo fluxo pode ser vista como um grafo de dependências de informações. Este grafo pode ser utilizado por vários métodos de diagnóstico diferentes [Nikovski, 2000, Przytula and Thompson, 2000, Jagt, 2002]. Se optarmos por uma abordagem estatística genérica, uma rede Baysiana é um método clássico, já amplamente estudado, que é adequado ao modelo. Um dos maiores problemas no uso de redes Baysianas nos processos de diagnósticos é a definição da estrutura de dependências da rede. No nosso caso a topologia de conexão entre *BFs* e *EDs* pode ser utilizada como definição inicial, além de possibilitar refinamentos ou detalhamento futuro. Nesta abordagem, os pontos de falhas considerados no diagnóstico são os *EDs* e os *BFs*. O projetista ou algum sistema de analise deve associar a cada destes pontos uma probabilidade inicial de falhas (Seções 6.5 e 6.6), juntamente com a probabilidade de detecção de falhas em cada teste realizado.

Os resultados dos testes de detecção de falhas são inseridos dinamicamente como eventos nos sistema de diagnóstico, como por exemplo, na rede Baysiana. O sistema de diagnóstico pode fornecer um índice de confiabilidade para os elementos chave do robô. Este índice deve ser utilizado no controle adaptativo. Uma abordagem básica para este índice é a probabilidade do fluxo gerar um valor para um atuador baseado em um valor proveniente de uma falha.

Um índice desempenho instantâneo (Seção 7.2.2) é calculado baseado em fatores definidos pelo projetista. Estes fatores podem estar associados com a missão, com a fase corrente e com valores do fluxo representados pelos *EDs* e *PCs* (Seção 6.4) usados no sistema. Os índices de desempenho e de confiabilidade podem ser combinados para fornecer um fator de ganho esperado para o robô. Este fator considera requisitos

internos e externos de desempenho e confiabilidade (Seção 7.2.3) sendo utilizado para seleção da melhor configuração pelo controle adaptativo.

O problema de adaptação passa a ser a seleção do fluxo, associado a uma determinada fase, mais adequado ao estado completo do sistema e ao meio percebido. O estado completo do robô pode ser definido por vários fatores, entre eles o estado de funcionamento ou falha dos elementos de hardware e software, requisitos de confiabilidade e requisitos de desempenho e qualquer tipo de influencia externa. O detalhamento das adaptações será realizado no Capítulo 8, que trata da implementação.

Utilizando as definições e atributos das missões e etapas e da topologia dos *BFs* e *EDs*, um grafo único de controle é gerado incluindo todas as transições de tarefas, de missão e de tratamento de falhas. Este grafo é posteriormente ampliado com informações sobre configurações equivalentes e utilizado como base do controle adaptativo. Todas as transições ou adaptações do sistema se tornam movimentações neste grafo de controle global chamado de Grafo de Controle Adaptativo (*GCA*).

Esta é uma visão geral do modelo desenvolvido, a qual será detalhada ao longo dos próximos capítulos.

Capítulo 6

Fluxo de processamento

> Nem tudo que pode ser contado conta, e nem tudo que realmente conta pode ser contado.
>
> Albert Einstein (1879 - 1955)

O fluxo de processamento definido para o modelo é responsável pela implementação de baixo nível do controle do sistema. Os constituintes deste fluxo e suas características são descritos neste capítulo. Os Blocos Funcionais e os Elementos de dados são detalhados explicitando as relações com os mecanismos de tolerância a falhas e o cálculo de confiança do sistema.

6.1 Blocos Funcionais – (*BFs*)

Os Blocos Funcionais *BFs* correspondem à abstração criada no modelo para um módulo de código desenvolvido pelo projetista, para processar toda a percepção e a decisão existentes no controle. A execução dos *BFs* realiza todo o controle de baixo nível. Os blocos funcionais correspondem a funções escritas pelo projetista na linguagem de programação[1] utilizada no controle, juntamente com estruturas descritivas que definem propriedades, atributos e interconexões existentes com os elementos de dados (*EDs*).

O controle, como já citado anteriormente, é realizado pela execução coordenada dos *BFs*, respeitando as dependências de dados existentes, e implementando um fluxo de processamento de dados, ou simplesmente fluxo de dados. Para tanto, é necessário se conhecer alguns atributos de processamento e a topologia de interconexão dos

[1] No protótipo desenvolvido a linguagem utilizada foi *C*.

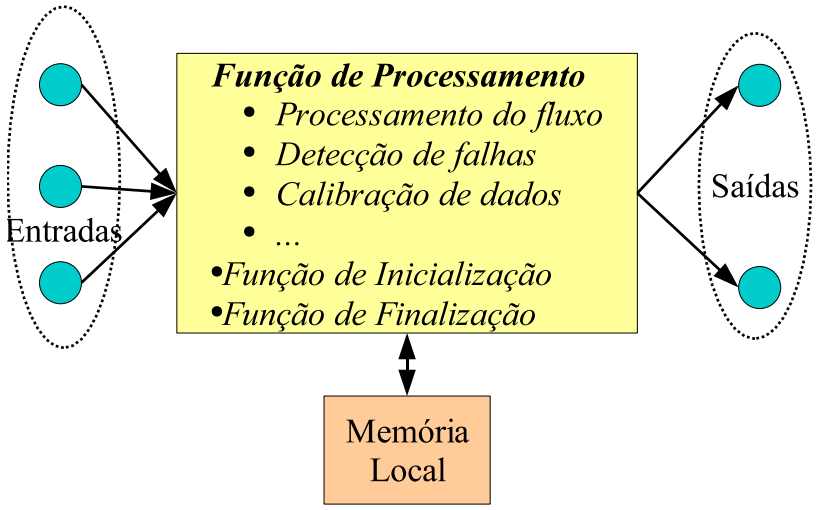


Figura 6.1: Elementos básicos de um Bloco Funcional *BF*

BFs e *EDs*. A plataforma de controle adaptativo *PCA*, que será mais bem descrita na Seção 8.2, realiza o escalonamento de *BFs* e fornece os recursos necessários para a execução do Fluxo de Dados, incluindo os processos internos de sincronismo e comunicação. A Figura 6.1 representa de forma gráfica a visão de um *BF* genérico. As suas estruturas descritivas são as seguintes:

Identificador - É nome do bloco funcional representado por uma *string*. Internamente o nome é mapeado em um identificador inteiro, o qual é utilizado na composição dos escalonamentos. Este identificador é utilizado na descrição do controle híbrido, nas parametrizações e nos arquivos históricos de execução (logs).

Descrição - É a descrição textual do processamento realizado ou funcionalidade específica do bloco funcional. Esta descrição só é utilizada em registros de eventos.

Natureza (opções: RT/USER/TA/TD) - Define a natureza de execução de um bloco funcional, especificando o mesmo é executado como um processo de tempo real ou como um processo em modo de usuário. As diferenças entre os *RT* e *USER* são descritas nas seções 6.1.2 e 6.2. Os *BFs* de natureza *TA* e *TD* são subclasses da natureza *USER* executando em modo de usuário. Os *BFs* de natureza *TA* realizam testes de aceitação na execução de um outro *BF*, enquanto os *BFsTD* realizam testes de detecção de falhas em Elementos de Dados *EDs*.

Os *BFsTA* e *TD* são blocos de execução opcional e têm o intuito de aumentar a confiabilidade do controle. Alguns dos atributos de cada *BF* variam em função da sua natureza, os quais são especificados na Seção 6.1.1.

***BFs* Incompatíveis** - Lista opcional com nomes de outros *BFs* os quais não podem ser executados no mesmo ciclo de controle. Este recurso é utilizado pelo projetista para limitar a composição automática dos escalonamentos identificando combinações de *BFs* impróprias.

***BFs* Associados** - Lista opcional com nomes de outros *BFs* os quais sempre devem ser executados no mesmo ciclo de controle. Este recurso é utilizado pelo projetista para limitar a composição automática dos escalonamentos identificando combinações de *BFs* necessárias.

Função de Processamento (*Process Function*) - A função de processamento é uma função em *C* responsável pelo processamento do *BF* propriamente dito. Em outras palavras é a função que realiza o trabalho do *BF*. Esta função acessa os *EDs* de entrada e os parâmetros de controle, realiza o processamento desejado e gera os valores para os *EDs* de saída. Os *EDs* estão presentes no código internamente através de constantes (*#DEFINE*) para acelerar todos os acessos a eles. Além disso, no *BF* normal é possível através de funções da *API* disponível consultar os identificadores das suas entradas e saídas. Este recurso é útil para permitir a reutilização da mesma função de código em mais de um *BF* na estrutura completa do fluxo. A funções da *API* disponíveis para programação dependem da natureza do *BF*, sendo esta restrição testada pela *PCA* com intuito de evitar erros.

Confiabilidade (*R*) - É a confiabilidade inicial ou estimada da Função de Processamento do *BF* em relação a falhas de software ou de processador. A probabilidade de falhas por erro no processador pode ser obtida pela probabilidade de falhas no processador e o tempo esperado de processamento da função. Já a probabilidade de falhas de software específica de uma função não é uma medida fácil de se obter. Podem-se utilizar métodos de análise específicos como os mostrados em [Bagchi et al., 1998], ou utilizar valores padrões para todas as funções.

***EDs* de entrada** - É o conjunto de *EDs* que são utilizados como fontes de dados pela Função de Processamento.

***EDs* de saída** - É o conjunto de *EDs* que são produzidos pela Função de Processamento.

Associação dos *EDs* de saída com os *EDs* de entrada - Os valores gerados para os *EDs* de saída de um *BF* podem não depender de todos os *EDs* de

entrada do bloco funcional. Neste caso, no modelo pode existir opcionalmente para cada *ED* de saída, a sua associação de dependência com um subconjunto dos *EDs* de entrada do *BF*. O *BF* do item (1) da Figura 6.2 na Página 64 exemplifica este conceito. Caso não seja refinada a estrutura de dependências, o modelo considera que todas as saídas dependem igualmente de todas as entradas, como é mostrado no item (2). Pode-se especificar que o *ED* $E4$ depende apenas dos *EDs* $E1$ e $E2$, enquanto o *ED* $E5$ depende apenas dos *EDs* $E2$ e $E3$. A informação das dependências existentes no fluxo pode ser utilizada no processo de diagnóstico, e quanto melhor representar a estrutura de interdependência real dos elementos físicos ou lógicos do robô, melhor será a qualidade do diagnóstico realizado.

Parâmetros de Controle - É o conjunto de parâmetros de controle *PCs* que são utilizados pela Função de Processamento. Pode existir associado aos parâmetros um atributo que indica a dependência dos custos de processamento do *BF* em função deste. Esta informação é considerada na avaliação das possíveis adaptações do sistema.

Memória Local - Define a área de memória **opcional** utilizada para armazenar o contexto local da função de processamento, entre as várias execuções. O acesso a esta área de memória é realizado através da *API*. Essencialmente, está memória é alocada e gerenciada pela *PCA* para permitir além da reexecução de um *BF* mantendo o contexto anterior no caso de falhas, a migração de processador em sistemas multiprocessados.

- Tamanho - É o tamanho da memória local manipulada pela *PCA* para o *BF*. .

Função de Inicialização - É uma função **opcional**, responsável por preencher os dados locais da função de processamento, ou realizar alguma outra configuração necessária ao sistema, como a ativação de algum hardware.

- Inicialização: [opções: Sistema/*BF*] - Este atributo define quando a função de inicialização deve ser executada, se uma vez junto com a inicialização do sistema, ou antes de cada execução do *BF*. No protótipo do sistema, só se considerou executar as funções de inicialização junto a inicialização do sistema, para simplificar o processo de ativação de um *BF*. Em sistema adaptativo, a opção de se ativar e desativar hardware ou software é importante, pois podem melhorar o aproveitamento de recursos escassos. Por exemplo, a desativação de um hardware pode reduzir o consumo das baterias do robô.

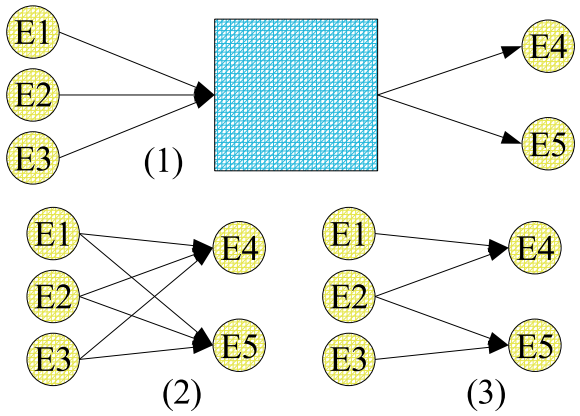


Figura 6.2: Exemplo de dependências entre entradas e saídas de um Bloco Funcional.

- Tempo Mínimo - É o tempo mínimo de espera entre a execução da Função de Inicialização e a Função de Processamento. Este tempo pode ser necessário no caso da ativação de algum hardware específico do sistema, que necessite uma espera para sua inicialização completa.

Função de Finalização - É uma função **opcional**, responsável por realizar alguma configuração necessária ao sistema, quando o *BF* for desabilitado. Esta função é executada na finalização do controle ou em um momento quando um *BF* será desabilitado temporariamente.

- Finalização [opções: Sistema/*BF*] - Opção de quando executar a função de Finalização. Se uma vez na finalização do sistema, ou depois de um período sem uso do *BF*.
- Tempo mínimo - Tempo de espera entre a ultima execução da Função de Processamento e a execução de forma automática da Função de Finalização.

6.1.1 Atributos específicos

Os *BFs* como elemento básico de processamento no fluxo podem ser de naturezas diferentes, como visto anteriormente. Cada bloco funcional de natureza diferente possui atributos próprios de acordo com sua finalidade. Estes são descritos a seguir:

- **RT** - Os *BFs* com natureza RT (*Real Time*) são processos cíclicos com restrições temporais rígidas. O atributo principal é o período de execução.

Período de Execução - Define a duração e variação permitida do ciclo de execução específico do processo de Tempo Real.

Prioridade - Define o nível de prioridade do processo identificado pelo *BF* em relação aos outros.

- **USER** - Os *BFs* com natureza USER são os blocos de software que efetivamente processam os dados no fluxo de dados. Este tipo de bloco pode conter internamente testes de detecção de falhas, entretanto, não são de execução opcional no fluxo. Se o teste interno existir, podem retornar um código de erro ao *PCA* caso seja detectada uma situação de falha. Um atributo próprio identifica a ação a ser realizada caso um erro seja detectado.

  - Probabilidade de Detecção de Falhas: Este atributo define a probabilidade de um teste realizado internamente, caso exista, de detectar uma falha, dado que esta ocorreu. Se não existir testes internos deste tipo, esta probabilidade é zero.
  - Probabilidade de Detecção de Falsas Falhas: Este atributo define a probabilidade de um teste realizado internamente detectar uma falha, dado que esta não ocorreu. Se não existir testes internos deste tipo, esta probabilidade é zero.
  - Ação de Recuperação de Falha: Corresponde a ação que a *PCA* deve realizar quando o *BF* USER retorna um código de erro. É importante ressaltar, que além da ação de recuperação, um evento de falha é sempre gerado e enviado ao sistema de diagnóstico, para que este seja capaz de alterar os valores de confiabilidade de cada uma das configurações. O teste realizado também pode informar a *PCA* qual foi o *BF* ou *ED* responsável pela falha ou responsável por gerar as informações detectadas como inválidas. Esta informação é disponibilizada para o processo de recuperação de falhas.

    Ignorar - A plataforma *PCA* simplesmente ignora o erro sem nenhuma ação corretiva específica.

    Re-execução - A Função de Processamento do *BF* é executada novamente com os mesmos valores nos *EDs* de entrada em com a Área de Armazenamento Local igual à primeira execução. A *PCA* define se a função será executada novamente no mesmo processador ou em outro diferente dependendo do hardware de processamento disponível, das restrições temporais do fluxo completo e da configuração instantânea do sistema. Além disso, a *PCA* é responsável por salvar a copia da área local do *BF* para permitir a recuperação completa do contexto.

    Função de Processamento Alternativa - Uma função de processamento alternativa é executada com intuito de recuperar da falha ou conduzir o

sistema a um estado mais seguro. Esta função alternativa deve possuir as mesmas saídas e utilizar um conjunto de *EDs* de entrada que já estão disponíveis no momento da falha.

Estado Seguro - A *PCA* realiza imediatamente uma troca da fase corrente em execução para uma fase de recuperação genérica associada em alto nível (Seção 7.1.4).

Reconfigurar - A *PCA* identifica se existe uma aresta de recuperação (Seção 7.4) associada ao teste corrente e tenta realizar imediatamente a reconfiguração específica definida para a recuperação da falha detectada. Caso não seja possível, a *PCA* seleciona e executa uma fase de recuperação genérica.

- **TA - Teste de Aceitação (*Accept Test*)** - Os *BFs* com natureza TA são os blocos de software cuja função é realizar testes de aceitação em relação a um *BF* USER que já realizou seu processamento no fluxo. O objetivo é detectar uma falha de software ou de processamento de forma a aumentar a confiabilidade do sistema como um todo. Retornam um código de erro referente ao *BF* testado. Este bloco é opcional podendo ou não ser executado em função da configuração do fluxo de processamento. Os atributos específicos são os seguintes:

  - *BF* Testado - Identificador do *BF* USER sob o qual o *BF* TA vai efetuar um teste de aceitação. Esta informação é importante por dois motivos: identificar o ponto onde a falha foi percebida; e identificar para a *PCA* a memória local do *BF* sob teste, caso esta necessite ser acessada.
  - Acesso à memória local [opções: Final / Final & Inicial] - Informação para a *PCA* controlar versões da memória local do *BF* sob teste. O teste de aceitação pode utilizar somente a imagem da memória após o processamento do *BF*, ou utilizar também uma imagem anterior à execução para facilitar as comparações e a detecção de estados incoerentes.
  - Probabilidade de Detecção de Falhas: Este atributo define a probabilidade do teste realizado detectar uma falha no *BF* analisado, dado que ela ocorreu.
  - Probabilidade de Detecção de Falsas Falhas: Este atributo define a probabilidade do teste realizado detectar uma falha no *BF* analisado, dado que ela não ocorreu.
  - Ação de Recuperação de Falha: Corresponde a ação que a *PCA* deve realizar quando uma falha no *BF* USER for detectada pelo *BF* TA. Como é relativo ao *BF* USER as opções são as mesmas já descritas.

- **TD - Teste de Dados** - Os *BFs* com natureza TD são os blocos de software cuja função é realizar testes de detecção de falhas sobre os valores dos *EDs* produzidos pelo fluxo de processamento. Estes *BFs* TD objetivam detectar valores incoerentes atribuídos aos *EDs* seja por fontes de dados redundantes ou pela mesma fonte ao longo do tempo. Além da funcionalidade de teste, um *BF* TD pode possuir também a capacidade de selecionar ou alterar dentro de um *ED* o valor que será lido ou propagado a partir dele no fluxo. Quando o *ED* possui redundância recebendo múltiplos valores, a execução do *BF* TD permite o uso de critérios mais elaborados como votações, médias de valores, ou o uso de qualquer outro método numérico para selecionar o valor mais adequado ao funcionamento do fluxo. Assim como o TA, este bloco é de execução opcional no fluxo.

  Na definição de um *BF* as suas entradas e saídas são completamente definidas estabelecendo de forma precisa a sua posição na topologia completa de interconexão. Esta característica não é adequada quando se deseja utilizar o mesmo *BF* TD em *EDs* distintos do fluxo. Como um *BF* TD é sempre associado a um *ED* foi escolhida uma solução foi composta em duas etapas: na primeira, o *BF* pode ter suas entradas e saídas parametrizadas na sua descrição através do uso de Identificadores Genéricos [2]; na segunda etapa, é incluída na descrição [3] do *ED*, no qual o teste será realizado, o mapeamento destes identificadores Genéricos com identificadores reais de *EDs* e *PCs*. A informação de mapeamento dos identificadores é inserida posteriormente como atributo do *BF* TD nos dados do escalonamento, permitindo desta forma a reutilização da função de processamento em múltiplos pontos no fluxo e a composição de uma biblioteca de funções de teste reutilizáveis que podem ser parametrizadas através de *EDs* e *PCs*.

  - Probabilidade de Detecção de Falhas: Este atributo define a probabilidade do teste realizado detectar uma falha no *ED* analisado, dado que ela ocorreu.
  - Probabilidade de Detecção de Falsas Falhas: Este atributo define a probabilidade do teste realizado detectar uma falha no *ED* analisado, dado que ela não ocorreu.
  - Ação de Recuperação de Falha: Corresponde a ação que a *PCA* deve realizar quando o *BF* TD retorna um código de erro. É importante ressaltar,

[2] Os Identificadores Genéricos correspondem simplesmente a uma faixa predefinida de constantes reservadas para esta parametrização, de forma a não criar conflitos com a definição dos *EDs*. Na associação do *ED* com um *BF* é realizado o mapeamento dos Identificadores Genéricos com os identificadores reais dos *EDs* e *PCs*.

[3] Esta descrição é detalhada na Seção 6.3.1.

que além da ação de recuperação, um evento de falha é sempre gerado e enviado ao sistema de diagnóstico, para que este seja capaz de alterar os valores de confiabilidade de cada uma das configurações. É importante ressaltar que o controle adaptativo pode selecionar outra configuração que apresente uma confiabilidade melhor para o estado do sistema. Assim sendo, o envio de eventos de falhas para o sistema de diagnóstico provoca, de forma indireta, ações de recuperação ou isolamento de falhas, que vão ocorrer com um atraso maior.

Ignorar - A plataforma *PCA* simplesmente ignora o erro sem nenhuma ação corretiva específica.

Estado Seguro - A *PCA* realiza imediatamente uma troca da fase corrente em execução para uma Fase Segura associada em alto nível. Este estado é definido para *PCA* no Grafo de Controle Adaptativo *GCA*.

Configuração Redundante - A *PCA* procura no Grafo de Controle Adaptativo *GCA* (Seção 8.1) uma aresta correspondente a recuperação da falha detectada pelo *BF* TD no *ED* analisado. Caso a aresta exista e for viável a reconfiguração selecionada, a *PCA* interrompe o fluxo corrente e inicia a nova seqüência de processamento de *BFs*. O novo fluxo iniciado deve conter redundância suficiente para tolerar a falha detectada. Caso uma reconfiguração adequada viável não seja encontrada, a *PCA* seleciona uma fase de recuperação genérica definida em alto nível para prosseguir com a execução do fluxo.

Adaptação - O erro força no final do processamento do ciclo o recalculo da confiabilidade das configurações e a possível seleção de uma nova adaptação.

A programação das funções associadas aos *BFs* é realizada como uma função normal da linguagem *C* que utiliza *API* de programação disponível para o acesso aos *EDs* e aos *PCs*. Além disso, a *API* fornece os recursos necessários para manipulação da área local de memória do *BF* e o acesso a outros recursos do *PCA*, como por exemplo, o modo corrente de funcionamento. O funcionamento interno as funções que acessam os *EDs* será detalhado na Seção 6.3. Os parâmetros de controle (*PCs*), utilizados na programação das funções presentes nos *BFs*, servem para ajustes automáticos e adaptações durante a execução, e serão detalhados na Seção 6.4. Toda a comunicação ou fluxo de dados entre os próprios *BFs* e a *PCA* deve ser realizada utilizando sempre as primitivas da *API*. Existem vários motivos para esta forte restrição, os quais foram descritos a seguir:

1. A abstração dos processos de comunicação permite a implementação e execução do sistema em arquiteturas variadas [Bagchi et al., 1998], multiprocessadas ou

não. A comunicação e o sincronismo entre os *BFs* podem ser implementados utilizando-se camadas de software específicas para a arquitetura alvo. Este tipo de abordagem oferece ao sistema, a possibilidade de alocação do processamento dos *BFs* em múltiplos processadores, o que é recurso essencial para implementação de tolerância à falhas de processadores, além de aumentar também as possibilidades de adaptação existentes. A abstração dos processos de sincronismo e comunicação permite que a plataforma de hardware possa ser ampliada mantendo o mesmo modelo de descrição e controle. É claro que a Plataforma de Controle Adaptativo *PCA* deve ser desenvolvida especificamente para a arquitetura alvo, e se esta for multiprocessada, deve-se implementar a camada de comunicação com recursos adequados de tolerância a falhas.

2. Utilizando sempre a *API*, o contexto de um *BF* passa a ser o seu conjunto de entradas (*EDs*), os parâmetros lidos e alterados (*PCs*), e a sua área de memória local. Se todo este contexto é controlável pela plataforma de software *PCA*, é possível permitir um gerenciamento automático de pontos de salvamento de contexto ou pontos seguros ([Randell and Xu, 1995]) (*Safe Points*) e a execução de processos de recuperação (*Roll Back*), no quais os *BFs* podem ser executados novamente a partir dos dados iniciais. Esta característica permite inclusive a recuperação em sistemas multiprocessados, nos quais podem existir as migrações de processo entre os processadores.

3. Como o contexto de um *BF* pode ser salvo, passa a ser possível isolar a sua execução em um modo de simulação ou teste, sem que o seu processamento e seus resultados perturbem o funcionamento normal do sistema. Este tipo de recurso pode ser utilizado no teste inicial para detecção de erros de software, ajuste de parâmetros, diagnóstico, ou inclusive na recuperação de falhas.

De forma sintética, a descrição do *BFs* contém a suas funções de processamento e toda a sua interface com o restante do sistema, além de processos de teste e probabilidades associadas às falhas. Com o uso um analisador de código (*Parser*) específico para o modelo é possível gerar automaticamente a interface de cada *BF* utilizando o código de suas funções de processamento.

A inclusão de "Testes de aceitação" específicos para as funções de "Inicialização" e "Finalização" não foram considerados no modelo pelos seguintes motivos: a freqüência de uso destas funções é muito inferior ao uso das funções de Processamento, assim a probabilidade de falhas por erros de transientes de processamento é menor; já é prática usual se inserir nas funções de inicialização ou finalização de um sistema, testes adicionais para validação do seu estado e das tarefas realizadas, o que não representaria um impacto significativo no desempenho do controle completo. Se o processo de inicialização ou finalização for muito complexo, estes devem corresponder

a *BFs* associados a fases específicas da missão em alto nível, e não apenas funções auxiliares associadas a funções de Processamento.

A estrutura dos *BFs* individualmente se assemelha aos objetos construtivos presentes em outras abordagens adaptativas para tolerância a falhas [Bagchi et al., 1998, Hecht et al., 2000]. Entretanto, na maioria destes sistemas, as políticas de redundância são criadas apenas com a replicação seletiva de processos de controle. A abordagem desenvolvida no nosso modelo, que abrange o uso de um fluxo de processamento e a inteligência presente nos *EDs*, facilita a implementação da tolerância a falhas adaptativa com variações maiores na estrutura de processamento que simplesmente o controle de cópias redundantes de processos. Trata-se da mesma forma uma reconfiguração para tolerar falhas de software ou uma falha de percepção ou atuação.

6.1.2 Blocos funcionais de Tempo Real

No modelo proposto os blocos funcionais foram divididos em dois grandes grupos, os *BFs* de Tempo Real (BFs^{RT}) e os *BFs* USER TA TD que executam como processos em modo de usuário (BFs^{U} *User mode*). A função principal dos BFs^{RT} é realizar a interface do controle com os elementos do hardware propriamente dito, coletando informações dos sensores, atribuindo-as aos *EDs*, recebendo informações dos *EDs* e enviando-as para os atuadores. Alguns detalhes deste processo de transferência serão vistos na Seção 6.2.

Vale a pena ressaltar, que no modelo proposto não foi feito tratamento específico no sentido de implementar a tolerância a falhas de software ou de processamento internamente aos BFs^{RT}, apesar disto, o modelo desenvolvido é compatível com soluções de tolerância a falhas adaptativa já existentes [Bagchi et al., 1998, OLIVEIRA et al., 2003]. A implementação de tolerância a falhas em processos de tempo real, como foi visto na Seção 2.4 se baseia no uso de políticas de replicação dos processos críticos e de testes de aceitação específicos, que permitem a recuperação em tempo hábil, mesmo existindo restrições rígidas.

No protótipo desenvolvido não foi necessário implementar processos de tempo real devido às características da arquitetura do *Nomad 200*, explicadas no Capítulo 9. Mesmo assim, os BFs^{RT} são processos de natureza completamente diferentes, que possuem um conjunto de considerações e características especiais para o modelo desenvolvido.

1. Os BFs^{RT} executam de forma periódica, possuindo freqüências mínimas e máximas, duração e prioridades próprias. Uma abordagem simplista que pode ser utilizada no modelo, é considerar o processamento demandado pelo conjunto de processos de tempo real como uma fração fixa do processamento disponível no sistema. Esta simplificação é razoável se mantiver no sistema as seguintes propriedades:

O conjunto de processos de tempo real em execução e suas propriedades não se alteram ao longo do tempo. Esta premissa não se mantém em sistemas onde a ativação e desativação de elementos hardware é possível e necessária para economia de recursos, e provoca a ativação e desativação de processos de tempo real associados ao hardware. Mas é razoável também considerar esta premissa válida quando não houver um ganho significativo em uma reconfiguração dos processos de tempo real.

A duração da execução de cada ativação de um processo de tempo real é muito pequena em relação aos processos que executam em modo de usuário. Isto normalmente é verdade em qualquer sistema, pois os projetos são desenvolvidos com o intuito de reduzir ao máximo os códigos de tempo real, que além de mais complexos, costumam ser pontos críticos.

A freqüência de execução dos processos de tempo real costuma ser muito superior aos ciclos de decisão implementados em sistemas complexos como robôs, pois são determinados por características dos sensores e atuadores.

2. Vários *BFs* de modo usuário podem gerar o valor para o mesmo *ED*, enquanto, apenas um e somente um BF^{RT} pode gerar um valor de um *ED*. Em outras palavras, quando o valor de um *ED* é gerado por um BF^{RT}, este é único. Não é permitida redundância explícita nos processos de tempo real. Caso esta exista, deve ser tratada pelo fluxo de processamento utilizando *EDs* diferentes.

3. Todos os *BFs* independentes da natureza utilizam conjuntos de primitivas específicas disponíveis na *API*, a qual se divide em dois conjuntos de funções para comportar todos os tipos de acesso dos processos de tempo real e os processos em modo de usuário. Embora as funções para as duas classes de *BFs* tenham interfaces equivalentes, existem algumas importantes diferenças internas:

   Os testes de detecção de falhas de um valor de um *ED* gerado por um BFs^{RT} é feita quando o valor é acessado para leitura por um *BF* (USER) que executa em modo de usuário. Esta escolha visa minimizar a manipulação de valores dentro da *API* utilizada pelos processos de tempo real.

4. Os *EDs* acessados por BFs^{RT} e *BFs* em modo usuário utilizam imagens diferentes do valor armazenado para facilitar e tornar mais eficiente o processo de sincronismo entre eles. Este sincronismo vai ser explicado na Seção 6.3.4.

Funções Básicas da *API* Oferecida Um conjunto de funções foi desenvolvido para permitir a interface com a Plataforma de Controle Adaptativo, juntamente com a implementação da comunicação do fluxo, acessos aos Parâmetros de Controle *PCs*. Algumas destas funções são mostradas na Tabela 6.1.

Acessam o valor dos *EDs*		
Função	Descrição	*BF*
*type*_GetValue(ED_{id})	Lê o valor armazenado no *ED*	BF^U
*type*_SetValue(ED_{id},*value*)	Armazena um valor no *ED*	BF^U
*type*_GetValueRT(ED_{id})	Lê o valor armazenado no *ED*	BF^{RT}
*type*_SetValueRT(ED_{id},*value*)	Armazena um valor no *ED*	BF^{RT}
Funções auxiliares		
GetGID(*ED* Genérico)	Retorna o identificador de um *ED* ou *PC* pelo identificador genérico. Esta função é utilizada para facilitar o múltiplo uso de *BFs*.	BF^U
Funções de acesso a *PCA*		
GetSyncTimeSensor(n)	Retorna o *timestamp* do sincronismo com os processos de tempo real n ciclos atrás realizado com os *EDs* associados a sensores.	BF^U
GetSyncTimeAct(n)	Retorna o *timestamp* do sincronismo com os processos de tempo real n ciclos atrás realizado com os *EDs* associados a Atuadores.	BF^U
ReturnTestResult(double)	Retorna um valor entre [0.0 e 1.0] correspondendo ao resultado do teste realizado. É utilizado no processo de recalibração.	BF^U

Tabela 6.1: Exemplo de funções da *API* utilizadas pelos *BFs*.

6.2 Controle de processamento

As arquiteturas de controle de robôs analisadas apresentaram como ponto comum algum tipo de ciclo envolvendo a percepção, uma etapa de decisão e a ação. Embora este ciclo esteja presente em várias arquiteturas de controle diferentes, existem muitas variações significativas. Nos controles reativos, a etapa de decisão é simples e rápida, e praticamente inexistente, em muitos casos pode ser uma função simples de mapeamento da percepção na ação. Nas abordagens deliberativas o processo de decisão pode ser extremamente longo quando se cria um mapa complexo do ambiente.

O modelo de implementação foi desenvolvido com o objetivo de ser capaz de aceitar as abordagens diferentes, e, além disso, facilitar a implementação de tolerância a falhas adaptativa. Quando estudamos os métodos existentes de tolerância a falhas podemos destacar duas abordagens sobre o controle de processamento: superdimensionar os recursos de processamento do sistema para suportar sempre o pior caso, obtendo assim um controle que opera na situação normal com grande ociosidade no processamento e muitas vezes com desperdício de recursos; utilizar uma abordagem adaptativa, a qual concentra o uso dos recursos nas ações mais importantes a cada

momento para o sucesso da missão. Se as ações críticas de uma missão podem variar ao longo do tempo, a abordagem adaptativa oferece uma razão melhor entre o custo e o benefício, principalmente em sistemas móveis com limites construtivos para disponibilizar recursos.

A programação do controle de sistemas como os robôs, satélites e outros, envolve sempre a interação com o hardware e com o ambiente. Neste caso, requisitos rígidos de tempo são necessários, seja para amostrar um sensor ou para controlar a rotação de um motor. A programação obedecendo a critérios rígidos de tempo é conhecida por programação de Tempo Real.

O modelo desenvolvido se baseia em alterações de configurações do fluxo para se permitir às adaptações do sistema. Quando se varia a configuração de *BFs* no fluxo, se altera também o tempo de processamento deste. Quando o tempo de decisão é alterado, é alterado também o tempo de reação do robô aos eventos internos e externos. Os controles de sensores e atuadores possuem restrições rígidas de tempo ditadas muitas vezes pela estrutura ou pela física do sistema.

Para atender estes requisitos no modelo foi necessário separar os processos de decisão dos processos de controle de nível mais baixo dos atuadores e sensores. Essencialmente, os processos de tempo real crítico ("*Hard Real Time*") que controlam os atuadores e sensores são independentes do fluxo de processamento e interagem com ele através dos *EDs*. Estes processos utilizam o escalonador próprio do sistema operacional e acessam áreas críticas dos *EDs* protegidas por semáforos. O fluxo de processamento dos *BFs* é implementado por um escalonador embutido no *PCA* e funciona em modo de usuário. O fluxo passa a ter um tempo de processamento limite para se obter os dados dos *EDs* associadas à atuação, mas sem restrições tão rígidas.

A separação em dois tipos de processos visa reduzir a complexidade total do projeto. Isto porque a dificuldade de desenvolvimento é muito maior quando se trata de processos de tempo real. Principalmente quando se trata de testar o sistema para identificar falhas de software.

No modelo desenvolvido, os BFs^{RT} são processos ou tarefas cíclicas de tempo real que realizam a interface do controle com os elementos do hardware propriamente dito, coletando informações dos sensores e atribuindo-as aos *EDs* e recebendo informações dos *EDs* e as enviando para os atuadores.

A divisão em duas classes dos blocos funcionais objetiva minimizar o desenvolvimento de código de tempo real, além de oferecer uma interface padrão baseada nos *EDs* para comunicação entre os processos de natureza diferentes. Se o desenvolvimento dos BFs^{RT} for realizado utilizando uma *API* do modelo, o sincronismo pode ser realizado pela *PCA*, o que embute algumas seções críticas em funções compartilhadas e previamente testadas. Estes detalhes serão apresentados na Seção 6.1.2.

No modelo desenvolvido, esta estrutura de separação dos processos pela

rigidez nas restrições de tempo foi inspirada na arquitetura de processamento do RTLINUX [Barabanov, 1997] e do robô *Nomad* 200 [Nomadic, 1997a, Nomadic, 1997c].Nestas duas arquiteturas, fica evidente a diferenciação dos processos de tempo real crítico que controlam diretamente o hardware e os processos de decisão do controle, que obedecem a restrições de tempo não tão rígidas.

Na arquitetura do *RTLINUX* os processos são divididos em processos de tempo real (*real time mode*) e processos de usuário (*user mode*). Os processos de tempo real são periódicos com rígidas restrições de tempo e são executadas em um *microkernel* que possui um escalonador próprio. No *RTLINUX*, o próprio *kernel* normal do *Linux* é executado como uma tarefa de baixa prioridade, que pode ser interrompida, neste escalonador de tempo real (*Real-time Microkernel*). Os processos em modo de usuário, que não possuem restrições de tempo rígidas, utilizam o próprio escalonador do *Kernel* normal do *Linux*. Uma *API* específica é disponível para se criar e gerenciar processos que executam no *microkernel*, e permitir a comunicação com os processos que executam no *kernel* do *Linux* em modo de usuário. O *RTLINUX* possui uma estrutura que evidencia as diferenças entre as duas classes de processos e oferece uma plataforma atrativa para a implementação do modelo proposto.

O robô *Nomad 200* possui uma arquitetura multiprocessada, na qual, os processo de tempo real que controlam os sensores e os motores executam em processadores dedicados. A descrição mais detalhada do *Nomad 200* pode ser vista na Seção 9.1 na Página 185.

Os processos de tempo real (BFs^{RT}) executam portando de forma síncrona com períodos pré-determinados, enquanto o processamento da percepção e da decisão realizado pelo fluxo de *BFs* executa em modo de usuário com restrições menos rígidas de tempo. A execução de cada *BF* individualmente não é síncrona com os BFs^{RT}, mas a duração do fluxo completo deve respeitar limites de tempo estabelecidos pelo projetista entre a percepção e a ação associada. O período do ciclo de processamento do fluxo deve ser estabelecido de forma precisa. Normalmente o seu início é definido pelo sincronismo dos *EDs* associados aos sensores e o seu fim pelo sincronismo dos *EDs* associados aos atuadores.

Existe ainda a possibilidade dos BFs^{RT} relacionados com elementos de hardware específicos ou com naturezas físicas diferentes coletarem dados com freqüência muito diferentes, ou com um sincronismo diferente. O projetista pode desejar trabalhar em um determinado momento, com o conjunto de todos os dados perceptivos coletados o mais próximo possível no tempo, e em outros momentos pode desejar utilizar o dado de um sensor o mais recente possível juntamente com outros dados mais antigos. A solução adotada no modelo foi o agrupamento dos *EDs* que devem ser sincronizados simultaneamente. Em uma situação comum, todos os *EDs* associados às entradas são sincronizados no início do ciclo e ao final são sincronizados todos os *EDs* associados aos atuadores. Os pontos de sincronismo devem ser pré-determinados e foram embutidos

no fluxo como *BFs* especiais. Para garantir e aproveitar a ortogonalidade do sistema, o controle das transições de fase e das adaptações também foi incorporado ao fluxo utilizando *BFs* especiais, da mesma da mesma forma que o sincronismo.

Influencia da tolerância a falhas no controle de processamento: A tolerância à falhas adaptativa baseia-se no princípio de concentrar o uso dos recursos no ponto de maior ganho de acordo com o estado e objetivo correntes da missão. Com este intuito, pode-se imaginar que o tempo de decisão pode variar em função do objetivo corrente e da própria situação de falhas. A duração do ciclo de processamento do fluxo poderia ser definida de forma fixa como a maior duração necessária, entretanto esta abordagem proporcionaria uma grande perda de desempenho quando o sistema não estiver com defeitos.

No modelo optou-se por um ciclo de processamento do fluxo variável de acordo com a configuração corrente do sistema. Se o tempo de reação do robô é alterado, o seu controle deve ser ajustado apropriadamente à nova situação. Isto é feito basicamente com a indexação ou associação de valores de determinados Parâmetros de Controle *PCs* com a duração do ciclo ou com cada configuração específica. Simplificando, associado a cada configuração do controle, existe a definição do fluxo a ser executado, o seu período e quais são os parâmetros de controle adequados.

Existe ainda a possibilidade de a cada configuração diferente de fluxo associar atributos dos BFs^{RT} que proporcionaria uma melhor capacidade do controle de processamento. A alteração dinâmica dos processos de tempo real não altera a definição do modelo e sim a sua implementação, que passa a ser muito mais complexa, assim sendo foi deixado como trabalho futuro.

O controle de um robô pode envolver diversos níveis de complexidade nas decisões e consequentemente no processamento. Por exemplo, a escolha de uma determinada direção pode envolver a formação de um mapa elaborado do ambiente ou a descoberta da posição atual, o que pode consumir muito tempo. A detecção de um obstáculo na direção atual de um robô em movimento deve obriga-o a parar ou a desviar o mais rápido possível. Estes dois exemplos envolvem a direção do robô, mas podem operar em períodos de tempo muito diferentes. Uma possibilidade também analisada no trabalho foi à coexistência destes ciclos simultâneos ou concorrentes com durações distintas, muitas vezes com ordens de grandeza diferentes [Birk and Kenn, 2001].

O processamento de decisão do modelo foi montado como um fluxo de dados, sendo assim, a definição coerente de um ciclo é a execução completa do fluxo, a qual foi definida pelo sincronismo de percepção e ação efetuados nos *EDs*. Como a *PCA* controla a comunicação entre os *BFs*, esta pode criar múltiplas versões de dados nos *EDs* permitindo execuções paralelas e independentes dos fluxos e possibilitando sincronismo independente com os processos de tempo real. Resumindo, para se aceitar

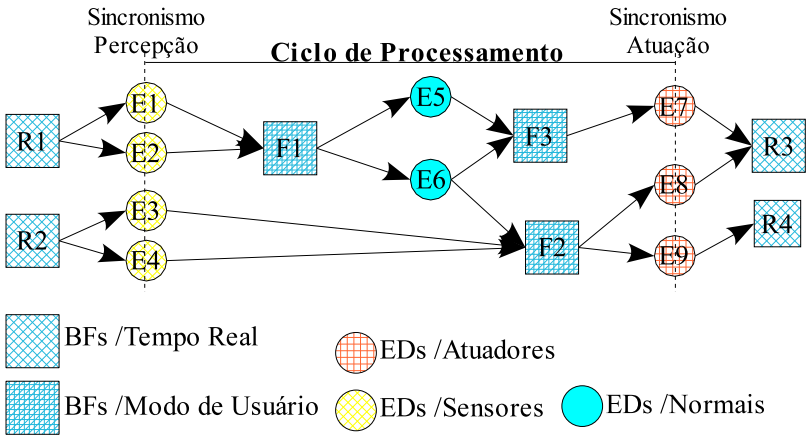


Figura 6.3: Definição básica de um ciclo de controle.

múltiplos ciclos ou fluxos simultâneos no modelo, é necessário incluir restrições temporais e prioridades no acesso dos *EDs* compartilhados, que determinem claramente a transferência de dados entre os fluxos e os processos de tempo real. Embora a simplicidade dos *BFs* e *EDs* seja mantida, a implementação da comunicação e escalonamento da *PCA* e as funções da *API* utilizadas para programação ficam muito mais complexas.

6.2.1 Ciclo de execução do fluxo

Um ciclo de processamento, contendo a percepção, a decisão e a ação, foi definido como a execução completa do fluxo de *BFs*. O ciclo de execução do fluxo mais simples inicia com o sincronismo dos *EDs* associados aos sensores e termina com o sincronismo dos *EDs* associados aos atuadores do sistema, como mostrado na Figura 6.3. Esta seção descreve alguns detalhes do implementado pelo escalonador da *PCA*.

O sincronismo entre os BFs^{RT} e os *BFs* é realizado essencialmente com o uso de duas imagens do valor armazenado em um *ED*. Os BFs^{RT} atribuem os valores de sensores na imagem de tempo real, e os *BFs* do fluxo lêem da imagem em modo de usuário. O comportamento dos *EDs* associados a atuadores é simétrico. Um processo de sincronismo protegido por semáforos iguala as duas imagens quando desejado. Este processo será detalhado na Seção 6.3.4.

O ciclo e a duração do processamento do fluxo de dados de baixo nível são fatores básicos para determinar a reatividade do sistema. Assim sendo, o projetista pode determinar para cada fluxo dois tipos de grandeza: os tempos máximo e mínimo entre os pontos de sincronismo e as freqüências máximas de mínimas de execução

de cada fluxo. As restrições podem ser inviáveis para o processamento disponível, ou inviabilizar apenas algumas possíveis configurações. Os métodos para este tipo de analise já foram amplamente estudados para alocação de processos de tempo real [Fohler, 1997], devendo ser embutidas em um ambiente de desenvolvimento para o modelo. É importante ressaltar que a demanda de processamento pelos processos de tempo real também deve ser considerada. A adaptação do sistema, em função da variação na duração do ciclo de processamento, é discutida em detalhes na Seção 6.4.1.

A informação das entradas e saídas de um *BF* já existe internamente na sua programação. Assim sendo, para se executar um determinado fluxo basta determinar a seqüência adequada dos *BFs*. Para tanto, é necessário e suficiente determinar um escalonamento de *BFs* que atenda aos requisitos do fluxo desejado respeitando as dependências de dados existentes. Estes requisitos são descritos na Seção 6.2.2. Se o processo de sincronismo receber um conjunto de *EDs* como parâmetros e puder ser ativado pelo escalonador, alguns dos requisitos definidos para o modelo podem ser facilmente alcançados. O fluxo mostrado na figura Fig 6.3 pode ser representado pelo seguinte escalonamento: $Sync_{E^1,E^2,E^3,E^4}, F1, F2, F3, Sync_{E^7,E^8,E^9}$.

O sincronismo restrito ao início e fim do fluxo pode não atender aos requisitos do projetista. Para se acessar dados de percepção ou de atuação coletados em momentos distintos no tempo, é necessário incluir outros pontos de sincronismo durante a execução do fluxo de dados. Para tanto, os *EDs* de sensores e atuadores que necessitam de um sincronismo diferente devem ser agrupados. Se no exemplo anterior os *BFs* $F1$ e $F3$ demoram muito tempo e o projetista deseja que os valores utilizados dos *EDs* (E3,E4) por F2, sejam os mais recentes possíveis. Neste caso é incluído no escalonamento outro ponto de sincronismo que vai atualizar os *EDs* com os últimos valores gerados pelas tarefas de tempo real. Na Figura 6.4 é exemplificada esta alteração. No escalonamento atual, foi incluído mais um estágio de sincronismo específico dos *EDs* ($E3$,$E4$) logo antes da execução do *BF* $F2$. Portanto correspondendo a: $SyncE^1, E^2, F1, F3, Sync_{E^3,E^4}, F2, Sync_{E^7,E^8,E^9}$.

A solução de incluir no escalonamento o sincronismo aumenta a liberdade do projetista na coleta e atuação do controle, entretanto não atende a possibilidade de ciclos com durações distintas. Imagine o fluxo da figura Fig 6.5, no qual em relação ao exemplo anterior, foi incluído mais um *BF* (F4), que gera o valor do *ED* ($E10$). As restrições de tempo do *BF* ($F4$) para se gerar o valor do *ED* ($E10$) são mais relaxadas, e o tempo para execução do *BF* ($F4$) é igual à duração do ciclo atual. Se $F4$ for inserido em todo ciclo, este terá sua duração dobrada e poderá não atender os requisitos de tempo da atuação ou reduzir diretamente o desempenho do sistema. Uma solução é permitir a execução de duas instâncias de fluxos distintas paralelamente e incluir restrições de precedência entre os processos de sincronismo. Se $F4$ for executado uma vez em cada quatro fluxos o impacto no processamento será de 25% de acréscimo na duração do ciclo.

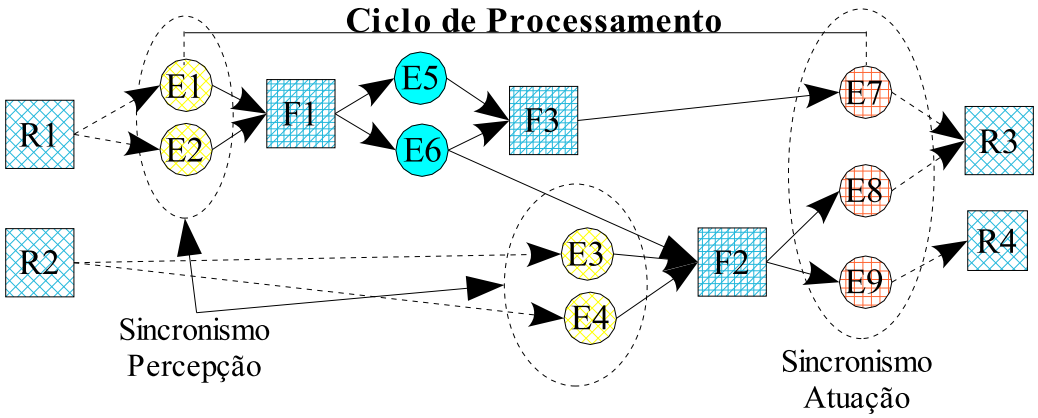


Figura 6.4: Definição de um ciclo de controle com sincronismo diferente nos sensores.

Escalonador 1: $Sync^1_{E^1,E^2,E^3,E^4}, F1, F2, F3, Sync^1_{E^7,E^8,E^9}, F4, Sync^1_{E^10}$.

Escalonador 2: $Sync^2_{E^1,E^2,E^3,E^4}, F1, F2, F3, Sync^2_{E^7,E^8,E^9}$.

Escalonador 2: $Sync^3_{E^1,E^2,E^3,E^4}, F1, F2, F3, Sync^3_{E^7,E^8,E^9}$.

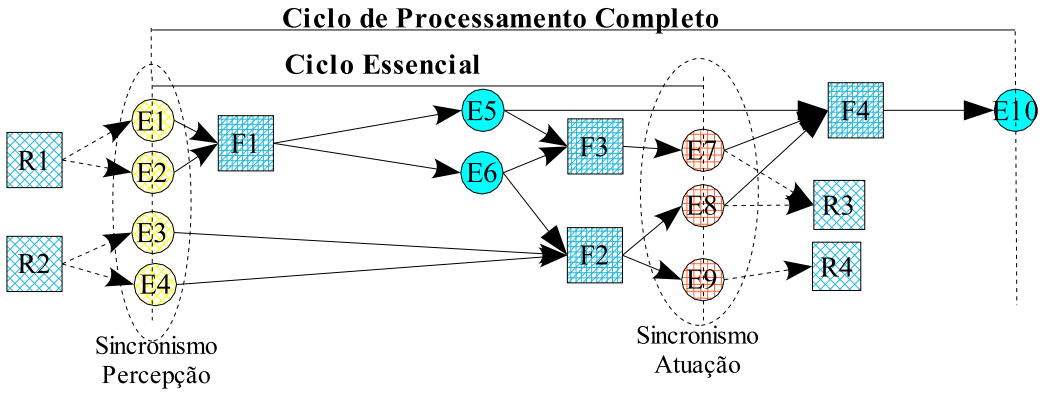
Escalonador 2: $Sync^4_{E^1,E^2,E^3,E^4}, F1, F2, F3, Sync^4_{E^7,E^8,E^9}$.

Precedência: $Sync^1_{E^1,E^2,E^3,E^4} < Sync^1_{E^7,E^8,E^9} < Sync^2_{E^1,E^2,E^3,E^4} < Sync^2_{E^7,E^8,E^9} < Sync^3_{E^1,E^2,E^3,E^4} < Sync^3_{E^7,E^8,E^9} < Sync^4_{E^1,E^2,E^3,E^4} < Sync^4_{E^7,E^8,E^9} < Sync^1_{E^10}$

Os requisitos principais de implementação para a execução de múltiplos fluxos simultaneamente são os seguintes:

- Um escalonador de *BFs*, com múltiplas instâncias, que seja capaz de executar dois ou mais ciclos simultaneamente, compartilhando contextos comuns e determinados pontos de sincronismo. O sincronismo é necessário para que os dados produzidos pelos fluxos sejam cronologicamente coerentes.

- Um controle da versão dos valores armazenados nos *EDs*, de forma a garantir a coerência dos dados processados pelo fluxo. O controle de versões é fácil de se implementar, pois todo o acesso aos *EDs* é realizado pela *API* disponível, a qual pode encapsular recursos de controle de versão de dados associados ao escalonador da *PCA*.



Escalonamento 1: Sync(E1,E2,E3,E4),F1,F2,F3,Sync(E7,E8,E9),**F4,Sync(E10)**

Escalonamento 2: Sync(E1,E2,E3,E4),F1,F2,F3,Sync(E7,E8,E9)

Figura 6.5: Definição de um fluxo de controle executado em múltiplos ciclos.

- O controle de prioridade entre as transições de fase da missão deve ser simples e bem definido. A seqüência de execução de testes de transição deve ser estabelecida, inclusive o comportamento dos fluxos já iniciados e não finalizados. No exemplo da Figura 6.5, suponha que existe uma transição baseada no *ED* (E7) e uma no *ED* (10). O teste baseado no *ED* (E7) pode ser realizado em todo ciclo menor, enquanto o teste baseado no *ED* (E10) só pode ser realizado no final de cada ciclo maior.

  $$Sync_{E^1,E^2,E^3,E^4}, F1, F2, F3, Sync_{E^7,E^8,E^9}, Trans_{E^7}, F4, Trans_{E^{10}}$$

  $$Sync_{E^1,E^2,E^3,E^4}, F1, F2, F3, Sync_{E^7,E^8,E^9}, Trans_{E^7}$$

- Uma maneira de especificar os seus requisitos de tempo. Uma opção é agrupando os *EDs* e definir restrições temporais entre os agrupamentos.

  $$(E1, E2, E3, E4) \rightarrow (E7, E8, E9) \Rightarrow (\triangle_{max}(T) = 1ms, Freq_{min} = 10Hz)$$

  $$(E1, E2, E3, E4) \rightarrow (E10) \Rightarrow (\triangle_{max}(T) = 3ms, Freq_{min} = 20Hz)$$

Concluindo, o modelo desenvolvido se mostra adequado para trabalhar com múltiplos pontos de sincronismo e múltiplos ciclos de processamento em paralelo devidos a dois fatores principais. O modelo de fluxo de dados já define de forma simples a integridade de um processamento serial, mesmo que existam múltiplas instâncias. As abstrações e o encapsulamento oferecido pelo *EDs* no fluxo facilita a gerência de múltiplas versões da mesma informação.

6.2.2 Seqüência de Execução dos *BFs*

A execução do fluxo de processamento, como já foi dito corresponde à execução seqüencial de um subconjunto dos *BFs* e de alguns processos especiais do sistema, como sincronismo e controle de transições. Para se definir a seqüência de *BFs* deve-se seguir alguns requisitos.

O critério mais claro é a precedência de dados do fluxo, ou seja, o valor de um *ED* deve ser produzido antes de ser consumido, mesmo que seja produzido por mais de um *BF*. Esta informação é contida na definição dos conjuntos de entradas e saídas de um *BF*. Além disso, no fluxo proposto não permitimos ciclos ($A \longrightarrow B$, $B \longrightarrow$A), justamente para garantir a existência de uma precedência bem definida. Com base nesta informação é possível garantir que sempre vai existir pelo menos uma ordenação parcial de *BFs*, capaz de executar o fluxo completo.

A existência de mais de uma ordenação parcial, é uma liberdade que pode ser utilizada como fator de otimização para outras características desejáveis. Um fator importante de um sistema tolerante a falhas é o tempo de recuperação, o qual está relacionado com o tempo de detecção de falhas. As falhas são detectadas por *BFs* específicos que provocam transições para recuperação de falhas. Assim sendo, a execução antecipada destes no fluxo pode melhor as características de tolerância a falhas no sistema.

Associado aos testes, existe no grafo de controle adaptativo uma aresta para uma configuração segura ou para uma configuração que recupere da falha detectada. O custo de cada adaptação ou recuperação de uma falha pode ser calculado previamente como a diferença de processamento entre os fluxos associados a cada configuração. Este custo pode ser utilizado como critério de prioridade entre os testes de um mesmo fluxo. Quanto maior o custo de recuperação, maior a prioridade dos testes associados na composição da seqüência de *BFs* do fluxo.

Também, com intuito de antecipar a detecção de falhas, pode-se tentar priorizar os testes realizados nos *EDs* com confiabilidade menor. Ou seja, testar primeiro o que é mais sujeito à falhas.

É importante ressaltar que nem sempre será possível adaptar o sistema em tempo hábil. Neste caso o sistema busca um estado de recuperação, o qual será detalhado na Seção 7.1.4. Portanto o grafo de controle só possui recuperações viáveis dentro das restrições de tempo existentes no sistema.

6.3 Elementos de Dados – (*EDs*)

> Espere o melhor, prepare-se
> para o pior e receba o que vier.
>
> Provérbio Chinês

6.3.1 Descrição Básica

Os Elementos de Dados, chamados de *EDs*, implementam a comunicação existente no fluxo de processamento responsável pelo controle de baixo nível, correspondendo a um fator chave no modelo desenvolvido. Essencialmente o *ED* é um identificador de um porto de comunicação genérico no fluxo de dados, utilizado internamente a programação dos Blocos Funcionais *BFs*. São acessíveis através de uma *API* específica de programação, com um tipo de dado explícito e uma constante como identificação. Exemplos das funções disponíveis são vistos na Tabela 6.1.

A estrutura de fluxo de processamento ou fluxo de dados se estrutura em função da forma ou seqüência em que as informações são processadas e utilizadas da percepção a atuação. Esta visão de fluxo de informações facilita a identificação da redundância de informações que existe ou pode vir a existir neste fluxo. Portanto favorece diretamente a implementação de alguns recursos de tolerância a falhas que se baseiam na existência e no uso de redundância de informações.

O uso dos *EDs* para comunicação, internamente ao código dos *BFs*, proporciona um nível de abstração para o projetista em relação a dois aspectos principais do fluxo. O primeiro, é que não é necessário se preocupar diretamente com a origem ou com o destino dos dados processados, podendo estar conectado tanto a outro elemento do fluxo ou a um processo de tempo real. O segundo é que pode existir multiplicidade de valores na entrada e na saída e isto não interfere diretamente com o código desenvolvido. Esta abstração, junto com outras características do modelo facilitam a implementação de múltiplas configurações, e consequentemente, a implementação de tolerância a falhas adaptativa.

Os *EDs* desempenham papel fundamental na implementação deste fluxo por se tratarem de conectores genéricos, capazes de receber informações simples ou múltiplas mantendo a mesma interface de programação. Essencialmente, permitem que as reconfigurações do fluxo sejam realizadas com a ativação e desativação dos blocos funcionais, sem a necessidade de adequações internas no código dos *BFs*. As adaptações são controladas externamente aos blocos básicos constituintes do processamento de baixo nível do controle. Esta abordagem oferece algumas vantagens:

1. Aumenta a confiabilidade do código executado no controle. As reconfigurações são externas aos *BFs*, tornando-os mais simples e com um número menor de estados internos. Esta redução no número de estados melhora a confiabilidade da programação, pois facilita a execução de testes funcionais e aumenta a cobertura dos mesmos. A razão entre as entradas do bloco e seus estados internos é aumentada, proporcionando assim um maior controle e possivelmente uma maior cobertura de falhas.

2. Facilita reutilizar blocos de código já implementados. Os blocos passam a ser mais simples e com funções bem definidas. Fica mais fácil reutilizá-los em

missões e fases diferentes do mesmo controle, ou em projetos completamente diferentes.

3. A simplificação dos *BFs* como reduz a diversidade dos caminhos internos, também reduz a variabilidade no seu tempo de execução. Este é um fator importante no conhecimento do tempo gasto pelo fluxo de processamento, pois aumenta precisão do mesmo.

Os *EDs* podem estar associados a elementos físicos como sensores ou atuadores, ou simplesmente associados a alguma informação do controle. A cada *ED* está associado um conjunto de atributos que definem o seu funcionamento. Os valores dos *EDs* são produzidos e consumidos pelos blocos funcionais através de funções da *API*. Estas manipulam os valores associados a cada elemento, e podem inclusive acessar e alterar os índices de confiabilidade. Os elementos de dados *EDs* são responsáveis pela conexão dos diversos blocos funcionais. Uma constante inteira foi utilizada para a identificação de cada *ED* do sistema pelas funções da *API* no intuito de tornar a interface o mais leve possível. Os elementos descritivos básicos de um *ED* do modelo são descritos a seguir:

Identificador – O identificador do *ED* é uma constante inteira representada por um *#DEFINE* da linguagem *C*. Esta escolha da forma do identificar foi feita para simplificar a seleção do *ED* na programação dos *BFs* com o intuito de reduzir o *overhead* introduzido pelo seu uso. Existe uma faixa de valores (Identificadores Genéricos) reservada para facilitar a generalização de *BFs*.

Nome É um texto utilizado nas mensagens e arquivos de registro e depuração para identificar o *ED* textualmente. Corresponde ao texto do próprio *DEFINE* utilizado nos arquivos de programa para identificar o *ED*.

Tipo do Dado Define o tipo da variável ou valor armazenado no *ED*. Os tipos válidos estão presentes na linguagem *C*. Quando o tipo for um ponteiro, é necessário definir em bytes o tamanho da área que deve ser alocada.

Tipos: *short, int, long int, double, char, char *...*

Tamanho: Parâmetro opcional definindo a área a ser alocada.

Domínio Define o domínio valido para os valores do *ED*. Este domínio depende do Tipo de Dado do *ED*. Pode ser utilizado, tanto para se saber quando os valores são aceitáveis em para testes de detecção de falhas, quanto para variar automaticamente entradas estimulando testes em um *BF* específico. Corresponde a um conjunto de faixas de valores aceitáveis, contínuas ou não.

Natureza do *ED* Corresponde à natureza do valor que é armazenado no *ED*, como por exemplo, se é associado a um sensor ou a um atuador.

Sensor O *ED* é associado a um sensor do hardware do sistema. Este atributo significa que o valor do *ED* será atribuído por apenas um BF^{RT} e que existe um processo de sincronismo para tornar o valor disponível para os *BFs* que executam em modo de usuário. As funções relacionadas com detecção de falhas ou calibração devem ser executadas antes que o *ED* seja acessado para leitura.

Atuador O *ED* é associado a um atuador do hardware do sistema. Este atributo significa que o valor do *ED* será atribuído por apenas um *BF* do fluxo e que existe um processo de sincronismo para tornar o valor disponível aos BFs^{RT}. As funções relacionadas com detecção de falhas ou calibração são executadas após o *ED* ser acessado para escrita por um *BF*.

Sistema O *ED* é associado com valores internos a *PCA*, tornando acessíveis aos *BFs* indicadores como a fase ou a missão corrente, estado de falha e outros.

Memória O *ED* tem o papel de implementar uma memória no fluxo de dados sem criar um ciclo de dependências. Cada *ED* de memória é associado a outro *ED* do sistema, sendo que, na transição de cada ciclo de processamento do fluxo, o valor do *ED* associado é copiado para o *ED* de memória. Em outras palavras, o valor de um *ED* de memória corresponde ao valor do ciclo anterior do *ED* associado. Este *ED* não pode ser saída de nenhum *BF* sendo exclusivamente manipulado pela *PCA*. É necessário também que se defina um valor default para o *ED* de memória para que não fique nulo na execução do primeiro ciclo.

ED Associado: O identificador do *ED* do qual o *ED* de memória copia o seu valor. O *ED* deve ser do mesmo Tipo de Dado e possuir o mesmo Domínio.

Valor Default: Valor default do *ED* de memória atribuído no primeiro ciclo de processamento.

Default É um *ED* utilizado apenas como recurso de programação, favorecendo a reutilização de funções dos *BFs*. Este *ED* é disponível apenas para leitura e sempre retorna um valor pré-determinado.

Valor Default: Valor default sempre fixo para o *ED*.

Normal Um *ED* normal se não enquadra na especificidade dos anteriores. O seu valor pode ser atribuído ou lido por múltiplos *BFs* do fluxo.

Confiabilidade – (r): Confiabilidade inicial do hardware associado ao *ED* quando este for um sensor ou atuador.

Função de Ajuste de Confiabilidade: Um elemento de hardware pode sofre influencias internas ou externas que alterem suas características de confiabilidade. Neste caso pode ser associado a um *ED* uma função que altere a confiabilidade associada, por exemplo, multiplicando um valor armazenado em um *PC*, o qual inclusive pode ser indexado. Existem vários exemplos para esta necessidade, como sensores infravermelhos que perdem sua confiabilidade na presença de radiação, ou um sensor visual que não funciona bem na presença de fumaça.

Número de Versões Número de versões dos valores de ciclos anteriores que são armazenadas no *ED*. O número de versões e os valores de outros ciclos podem ser acessados através de funções específicas da *API*. Este é um recurso importante para se detectar determinadas falhas, utilizando a variação do comportamento de um determinado valor. Por exemplo, um sensor de temperatura de um robô em ambiente normal variar entre duas leituras mais do que 20^o indica provavelmente uma falha do sensor. O número ideal de versões varia em função das características e da necessidade da informação.

***BFs* de Processamento** - Atributo opcional que define a existência de um ou mais *BFs* de natureza TD associados ao *ED*, para realizar funções de detecção de falhas ou ajustes do valor realizando calibrações específicas. As funções destes *BFs* como são direcionadas ao tratamento de dados simples ou básicos tendem a ser utilizada múltiplas vezes no mesmo controle. Assim sendo devem ser parametrizadas de acordo com este uso associado aos *EDs*. Estes blocos funcionais podem acessar todas as múltiplas versões do valor atribuída ao *ED* gerada em múltiplos ciclos ou geradas por caminhos diferentes no fluxo. Existem duas funções básicas para estes *BFs* associados:

1. *Teste* - Realiza algum tipo de teste com os valores armazenados no *ED*, retornando a *PCA* um código de erro adequado. Além disso, quando existir redundância pode selecionar que valor será propagado no fluxo.
2. *Ajuste de Valor* - Pode realizar funções de ajuste no valor armazenado no *ED*, permitindo, por exemplo, a calibração ou mudança de unidade de algum sensor ou atuador.

Para esta associação de um *BF* TD com um *ED* é necessário especificar um conjunto de parâmetros:

Identificador do *BF* - Identificador do *BF* responsável pela função desejada.

Parâmetros - Realiza um mapeamento entre os Identificadores Genéricos utilizados como entradas e saídas na descrição do *BF* TD com identificadores reais de *EDs* ou de *PCs* requeridos pela função de processamento.

IDENTIFICADOR_GENÉRICO $\longrightarrow$ IDENTIFICADOR_REAL[4]

EDCORRRENTE $\longrightarrow$ *EDVELTRANSLACAO*

PCGLIMITSUP $\longrightarrow$ *PCLIMSUPVELTRANSLACAO*

PCGLIMITINF $\longrightarrow$ *PCLIMINFVELTRANSLACAO*

Ordem - Ordem de execução deste *BF* em relação a outros que podem estar associados ao mesmo *ED*.

Objetivo - Essencialmente o objetivo define quando o *BF* é inserido no fluxo. Os possíveis objetivos são os seguintes:

Teste - O *BF* realiza um teste no valor do *ED* e pode ser considerado opcional. É executado para aumentar a confiabilidade de execução do fluxo.

Ajuste - O *BF* realiza um ajuste no valor do *ED* e pode ser considerado obrigatório sempre participando do fluxo quando o *ED* é utilizado para leitura.

Recalibração - O *BF* é utilizado em um processo automático de calibração. É uma função opcional que é executada normalmente antes de uma função de ajuste. É associado a outro *ED* posterior ao fluxo com uma função de teste associada, que quando detecta uma falha no *ED* associado, o sistema entra em uma configuração de recalibração.

ED + BF TD – O par composto pelo *ED* e pelo teste *BF* TD associado capaz de ativar o processo de recalibração. O *BF* que realiza o teste pode estar associado ao mesmo *ED* que possui o processo de recalibração. O teste informa a *PCA* a necessidade de recalibração através de uma função especifica da *API*.

Uma informação importante que deve ser incluída futuramente na descrição do modelo é um agrupamento de *EDs* os quais possuem testes de detecção de falhas associados e que devem ter seus testes ativados simultaneamente. Ou seja, os testes de todos os *EDs* do agrupamento são simultaneamente ativados ou desativados.

[4]O Identificador Genérico é utilizado internamente no código do *BF* e em sua descrição. A associação com o identificador real é realizada em tempo de execução através deste mapeamento armazenado nos dados do escalonamento de *BFs*.

6.3.2 Funcionamento de um *ED*

Um elemento de dados *ED* funciona essencialmente como um repositório de um valor para comunicação entre os blocos funcionais. Se o modelo for utilizado em um sistema multiprocessado, pode vir a implementar a comunicação mecanismos mais complexos como troca de mensagens ou memória compartilhada. Neste caso, a comunicação realizada pelo *EDs* deve embutir métodos de tolerância a falhas de comunicação adequados à arquitetura específica do sistema de processamento. Podemos citar um trabalho realizado no próprio departamento de Marcos Pego [OLIVEIRA et al., 2003].

O uso dos *EDs* oferece também uma abstração da plataforma de processamento utilizada. Isto simplifica muito o trabalho do projetista, permitindo o uso de plataformas de desenvolvimento diferentes da plataforma final de execução. Também facilita a execução de testes durante o desenvolvimento, com inserção de estímulos específicos diretamente em cada *BF* isoladamente.

Além da funcionalidade básica de comunicação, foram incorporados outros recursos importantes para facilitar a implementação da tolerância a falhas.Grande parte da possibilidade de redundância do modelo se baseia na possibilidade de um *ED* poder receber valores provenientes de *BFs* diferentes, como é mostrado na Figura 6.6. No item (1) o *ED* d^5 pode receber valores dos *BFs* $F1$ e $F2$ no mesmo ciclo. A questão passa a ser qual valor atribuído deve ser propagado para o restante do fluxo. A resposta mais simples é propagar o valor mais confiável. Para cada valor atribuído para um *ED* é calculado um valor de confiança que leva em conta o caminho de processamento a partir dos *EDs* iniciais associados ao hardware. A confiança nos valores do fluxo representa a confiabilidade de forma instantânea, sem considerar diretamente o fator tempo. O cálculo é realizado com base na probabilidade inicial de falhas fornecida pelo projetista, sendo esta atualizada pelo sistema de diagnóstico utilizado.

O calculo é feito dinamicamente pela *PCA* em função da topologia do fluxo de informações. A probabilidade de um *BF* produzir um valor resultante de uma falha é a probabilidade de uma ou mais entradas estiverem com falha ou houver uma falha de processamento do próprio *BF*. Considerando a probabilidade de falha f e pode-se calcular a confiança r sendo $r = 1 - f$ de que o valor é isento de falhas. Se a saída de um *BF* depende de todas as suas entradas, teremos a equação genérica Eq 6.3.1. No exemplo da Figura 6.6 a confiança $r_{F_1}^{d^5}$ do valor atribuído a d^5 por F_1 é mostrada na Equação 6.3.2, sendo r_{F_1} atributo de F_1 e r^{d^1} e r^{d^2} (confiança do *BF* e dos *EDs* de entrada). Apenas para reforçar, se os *EDs* forem associados a elementos de hardware, os valores de r são inicialmente fornecidos pelo projetista e atualizados pelo sistema de diagnóstico. Se no fluxo, estiverem ativos os *BFs* F_1 e F_2, a confiança propagada em r^{d^5} será a maior produzida como mostrado na Equação 6.3.4.

A *PCA* gera um valor de confiança progressivamente à medida que um *BF* acessa os seus *EDs* de entrada, criando um valor único para todas as suas saídas. Pode

ser que uma saída não dependa de todas as entradas do *BF*, sendo este caso definido junto à descrição do *BF* ou utilizando funções de controle de confiabilidade na *API*. As funções são necessárias devido à possibilidade de alterar as dependências dinamicamente em função de dados do fluxo.

$$r_{F_n}^{d^j} = r_{F_n} * \prod_{i \in F_n^{inputs}} r^{d^i} \tag{6.3.1}$$

$$r_{F_1}^{d^5} = r_{F_1} * r^{d^1} * r^{d^2} \tag{6.3.2}$$

$$r_{F_2}^{d^5} = r_{F_2} * r^{d^3} * r^{d^4} \tag{6.3.3}$$

$$r^{d5} = \max(r_{F_1}^{d^5}, r_{F_2}^{d^5}) \tag{6.3.4}$$

O processo de seleção do valor de maior confiança de um *ED* embute importantes recursos de recuperação de falhas. Primeiramente facilita a execução simultânea de caminhos redundantes no fluxo sem a necessidade de reconfiguração interna aos *BFs* como foi visto anteriormente. Em segundo, desprezando-se valores com uma confiança menor, se implementa o isolamento de falhas no acesso aos *EDs*. O controle deste isolamento de falhas, como já está embutido na *PCA* e no acesso aos *EDs*, é extremamente fácil de ser utilizado pelo projetista. É importante ressaltar que esta seleção depende do calculo de confiança, o qual depende do sistema de diagnóstico, responsável por atualizar a confiabilidade dos *EDs* e *BFs*. O diagnóstico depende dos testes de detecção de falhas realizados pelos próprios *BFs* do fluxo, como será visto na Seção 6.6.

A seleção de um valor pode ser realizada também por um *BF* TD que compare os dados fornecidos procurando incoerências. Podem ser realizadas algumas analises mais elaborada, percorrendo o histórico dos últimos valores ou comparando mais do que dois valores simultaneamente com o intuito de detectar a origem da incoerência. Se não for possível detectar a origem de uma falha, todos os elementos que contribuíram com a formação dos valores devem ter sua confiabilidade reduzida. Esta atitude pode reduzir a confiança completa do fluxo provocando uma adaptação ou a desistência da missão corrente. A existência de informações redundantes pode facilitar a identificação mais precisa do defeito e consequentemente, uma melhor adequação.

O exemplo da Figura 6.6 utiliza apenas dois caminhos redundantes. Neste caso pode ser impossível detectar a origem de alguma incoerência nos valores atribuídos ao *ED* d^5. A confiança de todas as entradas d^1, d^2, d^3, d^4 seria reduzida.

A confiança completa do fluxo pode ser dada pela confiança conjunta dos valores atribuídos aos *EDs* associados aos atuadores. Este assunto será detalhado na especificação da fase da missão, na Seção 6.5.1.

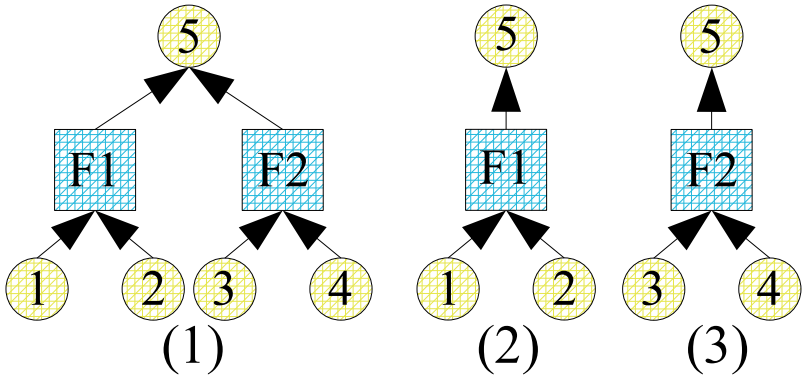


Figura 6.6: Exemplo de topologias redundantes.

Detecção de Falhas (*ED*)

A natureza ou origem do dado associado ao *ED*, principalmente se este for associado a algum hardware, pode variar muito e consequentemente o teste para detecção de falhas também varia. Pode ser necessário o desenvolvimento de testes muito específicos para um determinado projeto.

Os *BFs* TD associados aos *EDs* podem constituir uma biblioteca de fácil expansão por parte do projetista. As funções de detecção criadas podem utilizar uma interface de programação específica que fornece acesso aos dados internos dos *EDs* ou acessar diretamente as estruturas internas de armazenamento. A parametrização do acesso aos *PCs* e a outros *EDs* também simplifica o trabalho de personalização. A abordagem do uso de testes de detecção de falhas associados diretamente aos *EDs* possui alguns importantes pontos positivos em relação ao desenvolvimento do projeto do controle. Destacamos os seguintes:

1. Permite que o projetista amplie a biblioteca de métodos de detecção de falhas de maneira simples e padronizada, implementando novos métodos de detecção específicos. Uma vez desenvolvido um teste, este pode ser facilmente reutilizado no modelo.

2. Os métodos de detecção podem ser implementados de forma muito específica, permitindo a programação de maneira eficiente e reduzindo o custo de processamento associado.

3. O projetista não precisa se preocupar especificamente quando e como será chamado o seu método de teste, e sim quais os dados serão avaliados.

4. O uso de parâmetros específicos para a detecção de falhas, associados a cada *ED*, permite que o mesmo método seja utilizado simultaneamente em vários *EDs* diferentes no mesmo fluxo. Permitindo um alto grau de especificidade de cada teste, juntamente com o aproveitamento de código já implementado.

5. A detecção de falhas pode ser incluída ou removida do código muito facilmente, apenas alterando atributos associados ao *ED*.

A detecção de falhas no modelo sempre se baseia em um dos três tipos de redundância existentes: temporal, espacial e de informação. Os métodos descritos a seguir podem utilizar-se de mais de um tipo de redundância simultaneamente.

A redundância temporal é utilizada normalmente em computação com o processamento repetido de blocos de código com o intuito de detectar ou recuperar de falhas ocasionadas por panes no processador ou falhas transientes. No modelo proposto, basicamente qualquer *BF* pode ser re-executado pela *PCA*, inclusive com as versões diferentes, desde que não ultrapasse o tempo limite para o processamento do fluxo completo. Nos testes implementados nos *EDs*, além de métodos numéricos, pode ser utilizado qualquer critério de votação adequado para o uso com processamentos redundantes. Pode-se dizer que a flexibilidade oferecida pelo fluxo e pelos testes implementados através dos *BFs* TA e TD permitem o projetista utilizar a redundância temporal da forma que melhor lhe convier. Entretanto, como estes recursos dependem diretamente de implementação específica da *PCA*, não foram detalhados neste trabalho. Existem métodos genéricos já conhecidos e implementados em vários trabalhos [OLIVEIRA et al., 2003, Bagchi et al., 1998, Randell and Xu, 1995].

A redundância de informação existe, por exemplo, quando se conhece o domínio dos valores de um *ED*. Por exemplo, se existir um limite máximo e um mínimo definindo o domínio de um valor, é possível detectar uma falha em um *ED* quando o valor atribuído está fora do domínio determinado. O valor incoerente é facilmente detectado na Equação 6.3.5, identificando a presença de uma falha.

$$V_{Min}^{ED_y} \leq V_{BF_x t_n}^{ED_y} \leq V_{Max}^{ED_y} \tag{6.3.5}$$

Outro exemplo simples de redundância de informação, é se conhecer também a variação máxima do valor em relação à outra variável, por exemplo, o tempo. Se existir no *ED* informações históricas dos últimos valores processados, é possível verificar a diferença do valor atual em relação aos últimos valores obtidos; caso esta diferença seja incoerente com o máximo esperado, uma falha é detectada. Este tipo de teste é mostrado na Equação 6.3.6.

O conhecimento do funcionamento de qualquer módulo de um sistema corresponde à redundância de informação e muitas vezes pode ser empregada para se identificar incoerências entre o valor percebido e o esperado. Muitos testes desta natureza são comuns na analise de dados provenientes de sensores no campo de robótica.

$$abs(V^{ED_y}_{BF_x t_n} - V^{ED_y}_{BF_x t_{n-1}}) \leq f^{Step}(\Delta t, y) \tag{6.3.6}$$

Quando o controle esta sendo executado em uma plataforma multiprocessada, que tolere perda de processadores, pode-se dizer que existe redundância espacial para o processamento. Esta questão esta diretamente relacionada com a implementação da *PCA* e não com o modelo de descrição. Assim sendo não foi enfocada neste texto.

A redundância espacial se apresenta no controle de baixo nível do nosso modelo, como informações provenientes de fontes distintas no fluxo de processamento. É comum existir em um robô sensores iguais como múltiplos sonares ou sensores de natureza diferentes, mas capazes de fornecer informações correlatas entre si, como um sensor de velocidade e um de posição na mesma junta. Quando estas situações existem em um sistema, elas podem ser mapeadas no modelo através da produção do mesmo valor de um *ED* por mais de um *BF*. A Figura 6.7 e as Equações 6.3.7 e 6.3.8 exemplificam esta situação. Quando a diferença dos valores atribuídos ao mesmo ED^y provenientes das fontes redundantes (BF_x,BF_z) é maior que um determinado limite, é detectada a presença de uma falha. O limite utilizado na detecção de falhas pode ser uma função do *ED* (y) e das fontes de dados individualmente x e z, ou de outros fatores, permitindo inclusive um refinamento ou personalização do teste.

$$\exists V^{ED_y}_{BF_x t_n} \wedge \exists V^{ED_y}_{BF_z t_n} | x \neq z \tag{6.3.7}$$

$$abs(V^{ED_y}_{BF_x t_n} - V^{ED_y}_{BF_z t_n}) \leq f^{Err}(y, x, z) \tag{6.3.8}$$

Quando existem apenas duas fontes redundantes da mesma informação, normalmente pode-se detectar a presença de uma falha, mas é ser difícil realizar um diagnóstico adequado. A implementação do *ED* neste caso seleciona o dado proveniente da fonte mais confiável no dado instante. Quando existem mais do que duas fontes independentes, critérios mais elaborados podem ser utilizados, entre eles métodos clássicos como votações.

A redundância de informação pode ainda ser utilizada com modelos de comportamento muito mais complexos. Neste caso, são necessários métodos de análise elaborados ou a construção por parte do projetista de modelos de interação do robô com o meio. Utilizando o exemplo da Figura 6.10 na Página 113, a posição esperada de uma junta pode ser calculada como função da sua ultima posição, da sua velocidade, e do tempo decorrido entre as observações. O valor correspondente à posição esperada pode ser comparado com valores obtidos diretamente dos sensores da mesma. Caso os valores estejam incoerentes, é detectada uma falha. Este tipo de detecção de falhas no modelo pode ser também representada pela Equação 6.3.8. O que varia é a fonte da informação redundante, que pode ser proveniente de múltiplos sensores, ou de uma simulação de comportamento esperado, ou de ambos.

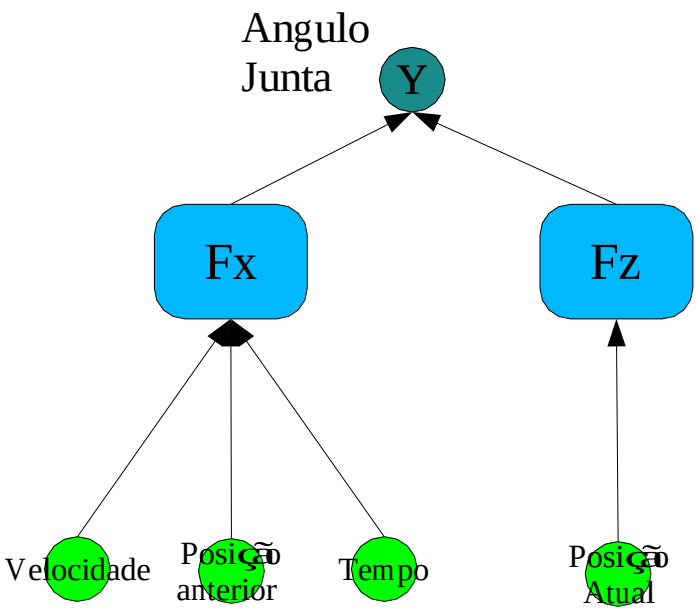


Figura 6.7: Teste de detecção de falhas comparando valores redundantes.

O modelo de comportamento de qualquer sistema é uma redundância de informação e pode ser utilizado para detectar falhas de funcionamento. Infelizmente, os sistemas robóticos possuem comportamento de difícil modelagem devido às incertezas inerentes aos seus elementos constituintes e a complexidade apresentada com a interação com o meio. No exemplo anterior, uma incoerência da posição da junta pode ser falha na atuação, ou falha nos sensores, ou simplesmente o manipulador encontrou um obstáculo em seu caminho. É responsabilidade do sistema de diagnóstico identificar o provável defeito.

Os métodos de detecção de falhas apresentados são básicos e apenas exemplificam as possibilidades do sistema, sendo que é possível implementar novos algoritmos e ampliar a biblioteca de *BFs* de teste disponíveis. Estes testes interagem com o sistema de diagnóstico através dos códigos de retorno de execução e das funções existentes na *API*.

Parâmetros de detecção de falhas

O objetivo da detecção de falhas é sempre identificar rapidamente e com precisão quando o funcionamento do sistema não é o esperado, indicando a presença de um ou mais defeitos. No caso de um robô móvel que interage com o ambiente, uma falha em um atuador pode provocar colisões, e significar prejuízos financeiros com a quebra do

equipamento, ou muito mais grave, com a possibilidade de ferir alguém. A detecção de falhas, como já visto, é um problema agravado nos robôs devido à incerteza inerente a suas partes mecânicas e a interação com o meio. Quando são utilizados em uma comparação para teste limites para os valores muito restritos ou estreitos, podem-se detectar falsas falhas. Quando se utilizam limites para os teste relaxados, pode-se demorar muito ou nunca detectar falhas reais.

Outro ponto importante a ser considerado, além da presença constante da incerteza, é que esta pode sofrer influência de outros fatores internos ou externos ao sistema. É desejável que os limites para detecção de falhas considerem as possíveis alterações, se adequando o melhor possível ao estado do sistema. Por exemplo, o erro máximo de posição de uma junta, pode sofrer influencia da velocidade máxima do atuador. A correlação de valores utilizados no teste juntamente com outros indicadores do sistema, pode levar a uma detecção de falhas mais apurada ou precisa e consequentemente influi diretamente na confiabilidade total do sistema.

Para atender o objetivo de utilizar parâmetros de detecção tão refinados quanto possível, é necessário que estes sejam ajustados individualmente para cada robô, e se fosse possível para cada estado do sistema e do meio. A individualização dos parâmetros de detecção pode ser realizada através de uma coleta de dados no próprio robô, quando este está em funcionamento considerado correto. Infelizmente, relacionar estes dados com o estado completo do sistema pode ser inviável, devido ao espaço de armazenamento necessário e ineficiência na recuperação do valor para os testes.

Portanto a melhor solução é relacionar os parâmetros de cada teste específico com subconjunto de indicadores do próprio sistema que influenciam diretamente nas medidas coletadas. Esta abordagem pode vir a melhorar significativamente a qualidade dos processos de detecção de falhas sem causar grandes impactos de perda no desempenho total do sistema.

Os parâmetros utilizados nos *BFs* de testes do modelo podem ser armazenados em um conjunto de Parâmetros de Controle *PCs*, que são descritos na Seção 6.4. Os *PCs* são parâmetros que podem ser multivalorados, cujo conteúdo é indexado pelo valor de outros *PCs* ou de algum *ED*. Esta flexibilidade permite uma maior grau de adaptabilidade do sistema, simplificando os ajustes internos que possam ser necessários dentro dos *BFs*.

No modelo, como os *EDs* podem ser considerados pontos de teste ou depuração, é possível exportar os valores processados no fluxo e por métodos matemáticos procurar automaticamente correlações entre eles. Estas correlações poderiam ser utilizadas para indexar as configurações. Entretanto, esta opção é deixada para os trabalhos futuros (Seção 10.1).

A solução definida para o modelo é relacionar a cada *ED* os *PCs* que vão ser utilizados como parâmetros nos testes e associar nos *PCs* os fatores que controlam a sua indexação. Portanto a definição das correlações utilizadas nos testes fica sob

responsabilidade do projetista do controle.

Os valores dos parâmetros de teste podem ser definidos também pelo projetista ou coletados a partir do funcionamento correto do sistema. As funções de teste implementadas pelos *BFs* podem ter dois modos de operação, os quais são controlados pela *PCA* e acessível ao *BF* através de uma função da *API*. Os modos são os seguintes:

TESTE -É o modo no qual os *BFs* TD executam o teste propriamente dito. Neste caso os *PCs* são utilizados como fontes de dados.

COLETA - É o modo no qual os *BFs* TD executam as comparações necessárias e geram os valores limites para os testes futuros. O robô é considerado em funcionamento correto. Neste caso os *PCs* são utilizados como fontes e destino dos parâmetros de teste gerados.

Um ponto importante é que se o *PC* utilizado vai ser indexado ou não é totalmente transparente para o código dos *BFs* TA. Esta independência facilita ao projetista testar várias opções de correlação entre os parâmetros de detecção de falhas antes de selecionar a situação final. Além disso, como os *PCs* são de controle da *PCA*, é muito fácil exportar e importar seus valores através de arquivos texto. Permitindo processos de ajustes e testes incrementais e alteráveis facilmente pelo projetista. Se o conjunto de fatores de influencia nos parâmetros de teste de um *ED* for reduzido, é possível que os parâmetros de detecção de falhas possam ser fruto de uma função ou tabela de mapeamento. A função de mapeamento deve considerar indicadores do sistema ou parâmetros de controle ou o valor de um outro *ED*, como é exemplificado nas Equações 6.3.9 e 6.3.10.

$$abs(V^{ED_y}_{BF_x t_n} - V^{ED_y}_{BF_x t_{n-1}}) \leq f^{Step}(\Delta t, y, PCs, EDs) \tag{6.3.9}$$

$$abs(V^{ED_y}_{BF_x t_n} - V^{ED_y}_{BF_z t_n}) \leq f^{Err}(y, x, z, PCs, EDs) \tag{6.3.10}$$

Resumindo, a estrutura de testes oferecida associada ao modelo proposto possui várias características atrativas, que facilitam a criação e reutilização de bibliotecas de funções prontas. Entre as características destacamos as seguintes:

- A padronização no uso dos diversos tipos de redundância.
- A capacidade de customização de um teste em função do ponto que é aplicado.
- Possibilidade de inserir gradualmente redundância e um número maior de testes aumentando de forma gradativa a confiabilidade do controle de baixo nível.
- A estrutura de parametrização facilita a coleta de valores para os testes e permite um grande refinamento nestes.
- Facilita a inserção de novos recursos de teste específicos para cada projeto.

6.3.3 Calibração em um *ED*

Um fator muito importante destacado na literatura é a necessidade de um sistema robótico, principalmente se tiver uma missão de longa duração, ser capaz de tolerar falhas de calibração. Os robôs possuem vários dispositivos como os sensores e atuadores que ao longo do tempo, ou sob a influência de agentes externos podem ter suas características alteradas. Embora sejam capazes de funcionar adequadamente, os valores gerados ou recebidos devem ser ajustados a sua nova realidade.

No modelo proposto, os sensores e atuadores estão associados aos *EDs*. Assim sendo, a solução proposta deve também estar. A estrutura se divide essencialmente em três partes e é mostrada na Figura 6.8.

1. Um *BF* TD (Adj) associado ao *ED* $D1$, por exemplo um sensor, possui o papel de ajustar o valor a ele atribuído em função de parâmetros armazenados em *PCs* ($P1$).

2. Um *BF* TD (Teste) associado a um *ED* $D2$que realizada teste com informações derivadas do *ED* $D1$. Este teste é associado a uma função de recalibração, e deve retornar pela função (*ReturnTestResult*) específica da *API* um valor continuo entre [0.0 ... 1.0] para indicar a presença de falhas na comparação realizada. Este índice é utilizado para guiar o processo de recalibração.

3. Um *BF* TD (Recalibra) também associado ao *ED* $D1$ que é ativado quando se deseja tentar uma recalibração. Este simplesmente altera os valores dos *PCs* da forma adequada. Se for obtida uma nova calibração adequada, o *BF* de recalibração é desativado e processamento volta ao normal. Se não, o *ED* continuará a apresentar valores errados e será considerado defeituoso.

A tentativa de recalibrar o parâmetro que controle o ajuste de um *ED* pode ser otimizada facilmente com múltiplas execuções do subfluxo entre o ponto de ajuste e o ponto de detecção da falha variando o parâmetro ajustado. O parâmetro de ajuste que provoca um erro com o menor índice é selecionado. Novos parâmetros são escolhidos e o processo é repetido até que o valor do *ED* seja ajustado corretamente ou que seja considerado efetivamente defeituoso. Desta forma, o problema de recalibração, pode ser resolvido no modelo basicamente com uma reconfiguração do fluxo e pequenas funcionalidades extras na *PCA*. A ativação desta reconfiguração pode ser feita com a recuperação de uma falha normal, através de arestas específicas no Grafo de Controle Adaptativo *GCA*. O detalhamento deste ponto fica como um trabalho futuro.

6.3.4 Sincronismo de tempo real

O sincronismo entre o fluxo de processamento e os processos de tempo real (BFs^{RT}) é um papel muito importante realizado pelos *EDs*. Pode-se dizer que existem dois

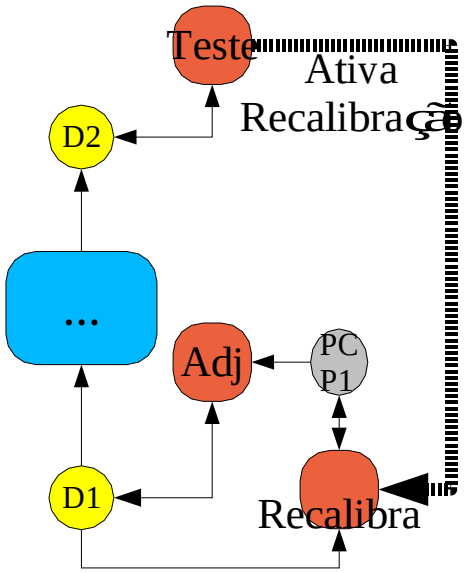


Figura 6.8: Processo de recalibração de um *ED*.

sincronismos diferentes:

1. Tornar disponível aos *BFs* que executam em modo de usuário os valores atribuídos aos *EDs* pelos BFs^{RT}. Estes *EDs* normalmente vão estar associados a sensores ou outras informações coletadas a partir do hardware.

2. Tornar disponível aos BFs^{RT} os valores atribuídos aos *EDs* pelos *BFs* que participam do fluxo de processamento. Estes *EDs* normalmente vão estar associados a motores ou outros atuadores de hardware.

O processo de sincronismo foi definido no modelo com um conjunto de objetivos, os quais são os seguintes:

- Isolar do projetista o problema de sincronismo entre os processos de tempo real e os processos de modo de usuário. Oferecendo uma interface de programação simples, testada e de fácil depuração.

- Torna o modelo mais independente da plataforma de execução. O sincronismo é implementado na *PCA* e na *API* oferecida ao projetista do controle do robô. Deve utilizar, portanto os recursos adequados de sincronismo para plataforma de execução, como semáforos, memória compartilhada, ou outros métodos de comunicação entre processos.

- Não criar contenção de nenhum tipo para os BFs^{RT} no acesso aos *EDs*. O acesso tanto de leitura, quanto de escrita nos *EDs* deve ter uma sessão crítica mínima. Além disso, deve respeitar os critérios de prioridade definidos entre os processos de tempo real.

- Atualizar os valores de vários *EDs* simultaneamente, garantindo que os *BFs* do fluxo e os BFs^{RT} utilizem conjuntos de dados compatíveis entre si, tanto os provenientes de sensores, quanto dos atuadores. Ou seja, realizar o sincronismo de vários *EDs* de forma atômica, sem gerar com isso contenção nos processos de tempo real.

Atender estes requisitos de sincronismo dentro da estrutura de *EDs* na forma que foram definidas as restrições de acesso não é uma tarefa muito complexa, como pode ser visto na Figura 6.9. Basicamente, deve-se trabalhar com três imagens do valor do *ED*. Uma imagem na área de processos de modo de usuário acessível aos *BFs* do fluxo, e as outras duas acessíveis aos BFs^{RT}. O exemplo mostrado se baseia no uso de semáforos e memória compartilhada. Entretanto, é possível se utilizar outros métodos de comunicação entre processos (IPC). No caso do RTLINUX, existe um conjunto de primitivas adequadas para esta comunicação. São diferentes do IPC tradicional, pois os processos de tempo real operam como módulos internos do *kernel*. No *Nomad 200* existem funções para atualizar um vetor de estados e comandos específicos de movimentação. Em outras palavras, a implementação do sincronismo vai sempre depender da plataforma de software e hardware utilizada, embora o conceito utilizado no modelo seja independente.

O sincronismo é realizado como parte do fluxo de processamento dos *BFs*, assim sendo, não é necessário preocupar com contenção ou concorrência nos acessos aos *EDs* por parte dos *BFs*. A utilização de uma imagem do valor do *ED* no modo de usuário é suficiente.

O acesso aos *EDs* pelo processo de tempo real é um pouco mais complexo, devido à concorrência existente com o processo de sincronismo, e a necessidade de tornar este processo atômico. Uma solução simples encontrada foi o uso de duas imagens do valor acessíveis pela *API* dos BFs^{RT}. Uma imagem é utilizada pelos BFs^{RT} enquanto a outra fica a disposição do processo de sincronismo para a transferência de dados com a imagem em modo de usuário.

A Figura 6.9 mostra o controle das imagens pelo processo de sincronismo. No caso dos *EDs* associados a sensores, um BF^{RT} a cada ciclo próprio, escreve valores na imagem corrente, por exemplo, na imagem I^0_{RT}. Quando o processo de sincronismo é iniciado, a imagem corrente é trocada para a I^1_{RT}. Os BFs^{RT} passam a acessar a imagem I^1_{RT}, enquanto os dados contidos na imagem I^0_{RT} são copiados para as imagens I_U acessíveis aos *BFs*. Quanto o sincronismo termina, a imagem acessível ao modo de usuário de todos os *EDs* atualizados possuem o valor da imagem I^0_{RT} no

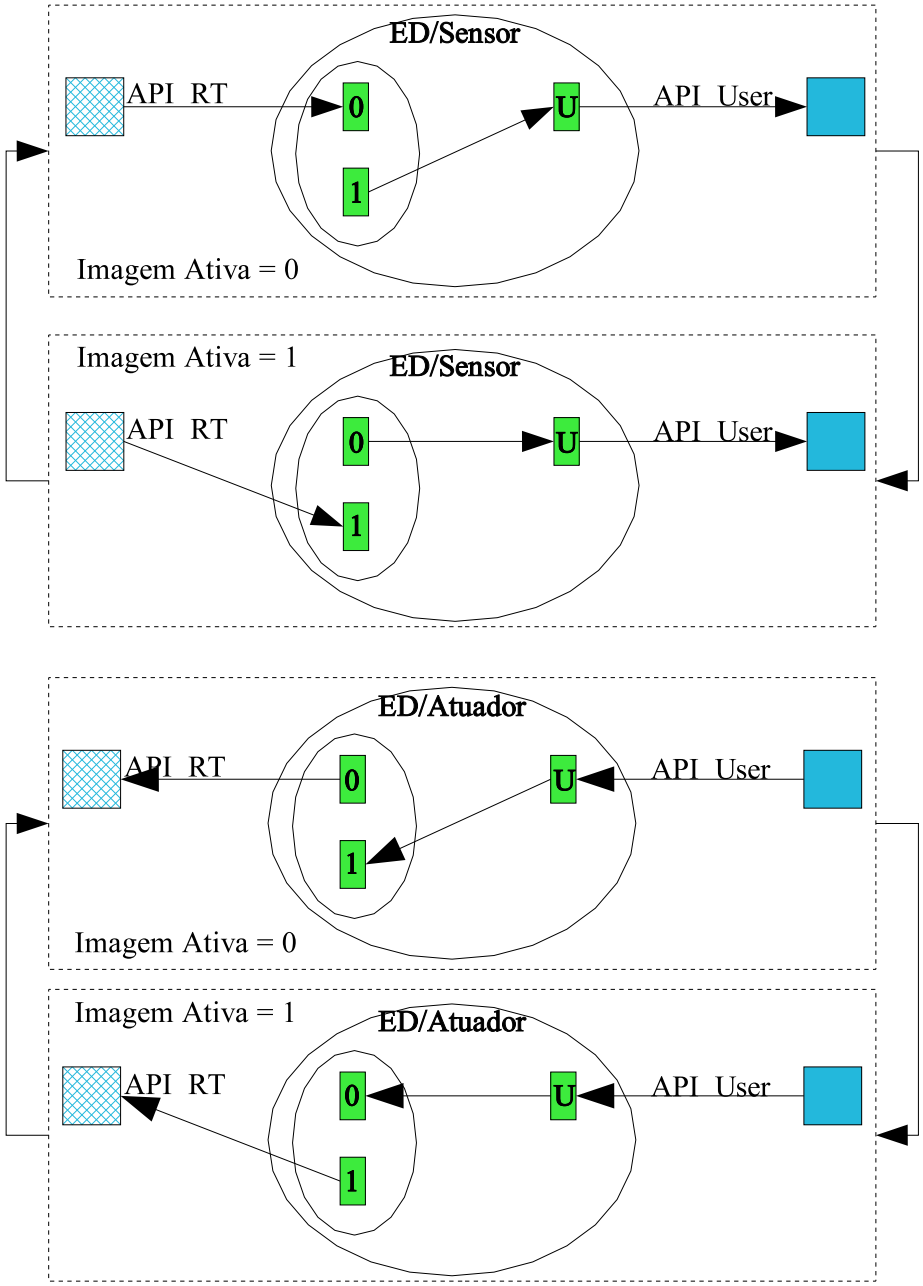


Figura 6.9: Processo de sincronismo de um *ED*.

momento da troca, obtendo desta forma uma operação atômica de sincronismo. Além disso, o sincronismo não gera contenção significativa para os processos de tempo real enquanto a copia de dados é realizada. O sincronismo dos valores nos *EDs* associados a atuadores é implementado de forma equivalente, sendo que a troca das imagens acessíveis aos BFs^{RT} é feita no final do processo, depois da cópia.

É importante ressaltar que foi considerado que o período de um BF^{RT} é muito inferior [5] ao ciclo de processamento do fluxo de *BFs* e consequentemente dos processos de sincronismo. Caso esta premissa não seja verdade, é necessário incluir na implementação alguns controles para garantir que não sejam utilizados dados desatualizados.

6.4 Parâmetros de Controle

O desenvolvimento de um sistema de controle para um robô, já é uma tarefa difícil. Se o projetista, para implementar a tolerância a falhas adaptativa, tiver que controlar simultaneamente várias configurações ou possibilidades diferentes no mesmo código, a tarefa fica mais árdua ainda. Os Parâmetros de controle *PCs* foram inseridos no modelo com o objetivo de facilitar determinadas tarefas do projetista, oferecendo uma interface para configuração e ajustes do processamento do controle de baixo nível que interaja adequadamente com o controle das adaptações. Essencialmente, os *PCs* são repositórios de valores acessíveis internamente aos *BFs* através da *API* desenvolvida, possuindo cada um, um identificador único e um tipo predeterminado. Alguns destes podem ser alterados automaticamente pela *PCA* em função de regras definidas pelo projetista, facilitando o processo de reconfiguração do fluxo de processamento.

Durante o estudo dos processos adaptativos e de tolerância a falhas, ficou clara a necessidade de facilitar a reconfiguração da atuação do sistema. Por exemplo, a perda de um conjunto de sensores pode exigir que o robô se movimente mais devagar para não provocar colisões. Ou reduzir também a mesma velocidade quando estiver carregando um objeto. Distribuir este tipo de controle espalhado no código dos *BFs* não seria uma tarefa simples. A solução encontrada foi criar um outro conjunto de elementos abstratos capazes de armazenar valores de configuração, de fácil acesso e manipulação. Estes elementos abstratos devem ser multivalorados e contendo parâmetros associados com estados internos ou externos. Ou até relacionados com uma fase específica da missão. Eles foram chamados de Parâmetros de Controle *PCs* e se mostraram adequados para várias opções diferentes, como armazenar configurações, parâmetros de detecção de falhas, ou valores que o projetista necessite. Os

[5] O período de um BF^{RT} deve ser no mínimo metade do período dos processos de sincronismo, garantindo que as imagens dos *EDs* acessíveis aos processos de tempo real vão ser atribuídas pelo menos uma vez, antes de serem copiadas.

atributos dos *PCs* são os seguintes:

Identificador O identificador do *PC* é uma constante inteira representada por um *#DEFINE* da linguagem *C*. Esta escolha da forma do identificar foi feita para simplificar a seleção do *PC* na programação dos *BFs* com o intuito de reduzir o *overhead* introduzido pelo seu uso. Assim como nos *EDs*, uma faixa predeterminada de identificadores foi reservada para criar os Identificadores Genéricos e facilitar a personalização dos *BFs* TD.

Nome É um texto utilizado nos arquivos de definição, nas mensagens e nos arquivos de registro e depuração para identificar o *PC*. Corresponde ao texto do próprio *DEFINE* utilizado nos arquivos de programa para identificar o *PC*.

Tipo do Dado Define o tipo da variável ou valor armazenado no *PC*. Os tipos válidos estão presentes na linguagem *C*. Quando o tipo for um ponteiro, é necessário definir em bytes o tamanho da área que deve ser alocada.

Tipos: *short, int, long int, double, char, char *...*

Tamanho: Parâmetro opcional definindo a área a ser alocada.

Domínio Define o domínio valido para os valores do *PC*. Este domínio depende do Tipo de Dado do *PC*. Pode ser utilizado para testes de depuração do sistema para variar estímulos em um *BF* específico. Corresponde a um conjunto de faixas de valores aceitáveis, contínuas ou não.

Classe De Atualização - Controla como o conteúdo do *PC* vai ser alterado pelo sistema.

Fixo: Possui um valor inicial que é atribuído na inicialização do sistema. Este valor não é alterado ao longo do tempo. É um tipo interessante para os parâmetros definidos pelo projetista.

Alterável: Possui um valor inicial que é atribuído na inicialização do sistema. Este valor pode ser alterado por *BFs* ao longo do tempo.

Persistente: Possui um valor inicial que é atribuído na inicialização do sistema. Este valor pode ser alterado por *BFs* ao longo do tempo. No final de uma execução do controle, o valor corrente sobrepõe o valor inicial anterior. Este é bem interessante para parâmetros de detecção de falhas calculados durante a execução do sistema.

Classe de Indexação - Controla se o valor do *PC* vai ser multivalorado e indexado e como é a forma de seleção.

Simples: O *PC* contém apenas um valor.

PC_{id}	Descrição	Tipo	Tam	Atualização	Classe
{P_TIMEOUT,	"Maximum timeout",	T_LONG,	0,	P_FIXO,	P_SIMPLES }
{P_ERROR_STR,	"Str error message",	T_ACHAR,	256,	P_FIXO ,	P_ SIMPLES}
{P_CYCLE,	"Max cycle",	T_LONG,	0,	P_FIXO,	P_AUTO }
{P_MAXVTRANS,	"Max translation vel ",	T_LONG,	0,	P_FIXO,	P_AUTO }
{P_MAXVSTEE,	"Max steering vel ",	T_LONG,	0,	P_FIXO,	P_AUTO }

Tabela 6.2: Exemplo de Parâmetros de Controle

PC_{id}	Default
{P_TIMEOUT,	(long)10L }
{P_ERROR_STR,	(char*)"Critical Error" }
{P_CYCLE,	(long)20L }
{P_MAXVTRANS,	(long)5 }
{P_MAXVSTEE,	(long)5 }

Tabela 6.3: Exemplo de inicialização dos Parâmetros de Controle

Automática: O *PC* contém um valor para cada configuração definida pelo projetista. Neste caso, pode-se dizer que o *PC* possui um vetor de valores, cujo índice de acesso é definido pelo nodo corrente do Grafo de Controle Adaptativo.

Indexada: O *PC* possui vários valores indexados por outros *PCs* ou por valores de *EDs* específicos, ou pela configuração definida pelo projetista.

As funções disponíveis na *API* oferecida para os *BFs* acessarem os valores nos *PCs* é a mesma independente da Classe de Atualização ou Classe de Indexação do *PC*. Existem ainda duas opções para as funções de escrita pelos *BFs* nestes *PCs*, controladas por parâmetros das funções de acesso:

Imediata : O valor escrito no *PC* é acessível imediatamente para ser lido por outros *BFs* no ciclo de execução corrente.

Sincronizada : O valor escrito no *PC* só é acessível aos outros *BFs* no próximo ciclo de processamento do fluxo. Se houver mais de uma escrita, fica valendo a última.

A Tabela 6.2 apresenta um exemplo com a definição de um conjunto de Parâmetros de Controle. Os *PCs* são referenciados na programação dos *BFs* através do identificador numérico, que foi atribuído pelo projetista. Os parâmetros de controle são inicializados com valores default, como visto no Exemplo 6.3, juntamente com a inicialização do sistema .

Os *PCs* com a Classe de Indexação *Simples* podem ser considerados como variáveis globais, ou como um recurso extra de comunicação entre os *BFs*, sem ter o custo, ou os controles associados ao uso dos *EDs*. Estes parâmetros possuem necessariamente um valor default atribuído na inicialização do sistema.

Os *PCs* com a Classe de Indexação *Automática* são utilizados no controle adaptativo, permitindo adequações do funcionamento e parametrização dos *BFs* em função de um estado global do sistema. Os *PCs***automáticos** são acessíveis pelos *BFs* da mesma forma que um da classe *Simples*, mas correspondem internamente a vetores de valores. O índice do valor que é acessado tanto para leitura quanto para escrita é definido internamente a *PCA* por informações contidas no nodo do Grafo de Controle Adaptativo (*GCA*).

A função principal dos *PCs* automáticos é oferecer ao sistema um grau a mais de liberdade ou flexibilidade para as adaptações. Os valores dos *PCs* devem refletir as adaptações do controle e do fluxo internamente nos *BFs*. A primeira utilidade que foi vista para se criar os *PCs* automáticos foi adequar as configurações internas dos *BFs* às alterações na duração ou freqüência de processamento do fluxo completo. As adaptações do sistema podem ter o custo de processamento muito diferentes, influenciando diretamente na duração do ciclo de controle. Pode ser necessário para a manter a coerência do sistema refletir variações no ciclo nos valores processados pelos *BFs*. Por exemplo, pode ser necessário para manter um grau de segurança do robô, diminuir a velocidade de um atuador quando o ciclo de controle aumenta. Ou aumentar a velocidade de um atuador, quando o ciclo é reduzido, podendo significar um aumento no desempenho do sistema (Seção 7.2.2).

No modelo desenvolvido, o projetista define um conjunto de valores para os *PCs* automáticos em torno de um Identificador de Configuração. No exemplo da tabela 6.4, o projetista definiu três configurações diferentes (*CONF01, CONF02, CONF03*). Depois é realizada uma associação entre a configuração, o *PC* automático e um valor inicial que deve ser atribuído.

Posteriormente, os Identificadores de Configuração específicos são associados com fases da missão ou diretamente a nodos do Grafo de Controle Adaptativo. Toda vez que o controle troca de estado, a configuração dos *PCs* automáticos é verificado, e caso seja necessário, o índice interno destes que seleciona o valor é alterado. Este processo é extremamente rápido, pois apenas um índice é corrigido.

Um grau maior ainda de refinamento é indexar os *PCs* por outros indicadores além do nodo corrente do *GCA*. Os *PCs* com a Classe *Indexada* são utilizados para um refinamento maior no controle adaptativo, permitindo adequações do funcionamento e parametrização dos *BFs* em função de qualquer indicador presente no sistema. Os *PCsIndexados* são acessíveis pelos *BFs* da mesma forma que um da classe *Simples*, mas correspondem internamente a vetores ou matrizes multidimensionais de valores. Os índices da célula que é acessada em um dado momento tanto para leitura quanto

P_MAXVTRANS:	index(Automatico)	
P_MAXVSTEE:	index(Automatico)	
$Config_{id}$	PC_{id}	Default
{CONF01,	P_MAXVTRANS,	(long)5L }
{CONF01,	P_MAXVSTEE,	(long)5L }
{CONF02,	P_MAXVTRANS,	(long)10L }
{CONF02,	P_MAXVSTEE,	(long)10L }
{CONF03,	P_MAXVTRANS,	(long)5L }
{CONF03,	P_MAXVSTEE,	(long)5L }

Tabela 6.4: Definição do conjunto de configurações válidas para os Parâmetros de Controle Automáticos

$Config_{id}$	PC_{id}/ED_{id}	Limite Inf.	Limite Sup.
{ IDCONFCL01,	EDSYS_CYCLE,	0 ms	10 ms }
{ IDCONFCL02,	EDSYS_CYCLE,	11 ms	20 ms }
{ IDCONFCL03,	EDSYS_CYCLE,	21 ms	30 ms }

Tabela 6.5: Definição de faixas discretas para valor de um *ED* ou *PC*.

para escrita é definida internamente ao *PC*. Os *PCs* indexados são mais genéricos na adequação que os automáticos por poderem sofrer influencias de qualquer indicador do sistema, inclusive as mesmas configurações que os automáticos.

6.4.1 Controle de indexação

Os elementos que controlam o índice de um *PC* podem ser tanto outros *PCs* quanto *EDs* do sistema ou do fluxo. Quando este elemento de controle possuir um domínio muito grande ou é uma variável real, pode ser inviável o seu uso. A solução encontrada no modelo é tornar discretos todos os valores utilizados para indexar os *PCs*. Deve ser definido para cada *PC* ou *ED* utilizado para indexar uma lista de faixas de valores ordenadas que correspondam a unidades discretas.

A solução é equivalente aos *PCs* automáticos, mas permite que a configuração corrente seja definida não apenas pelo nodo do *GCA*, mas por qualquer indicador do sistema, ou por um valor percebido. Imagine que a duração do fluxo de processamento acessível através de um *ED* do sistema e podem ser definidas faixas discretas como visto na Tabela 6.5.

Os identificadores discretos criados podem ser utilizados em um *PC* indexado. A lista com os elementos dos quais depende deve ser incluída na definição deste *PC*. Cada elemento corresponde a uma dimensão da matriz de valores interna ao *PC*. No exemplo da Tabela 6.6, os *PCs* P_MAXVTRANS e P_MAXVSTEE são indexados

P_MAXVTRANS:	index(Automatico, EDSYS_CYCLE)	
P_MAXVSTEE:	index(Automatico, EDSYS_CYCLE)	
$Config_{id}$	PC_{id}	Default
{CONF01, IDCONFCL01,	P_MAXVTRANS,	(long)5L }
{CONF01, IDCONFCL01,	P_MAXVSTEE,	(long)5L }
{CONF02, IDCONFCL01,	P_MAXVTRANS,	(long)10L }
{CONF02, IDCONFCL01,	P_MAXVSTEE,	(long)10L }
{CONF03, IDCONFCL01,	P_MAXVTRANS,	(long)5L }
{CONF03, IDCONFCL01,	P_MAXVSTEE,	(long)5L }
{CONF01, IDCONFCL02,	P_MAXVTRANS,	(long)5L }
{CONF01, IDCONFCL02,	P_MAXVSTEE,	(long)5L }
{CONF02, IDCONFCL02,	P_MAXVTRANS,	(long)10L }
{CONF02, IDCONFCL02,	P_MAXVSTEE,	(long)10L }
{CONF03, IDCONFCL02,	P_MAXVTRANS,	(long)5L }
{CONF03, IDCONFCL02,	P_MAXVSTEE,	(long)5L }
{CONF01, IDCONFCL03,	P_MAXVTRANS,	(long)5L }
{CONF01, IDCONFCL03,	P_MAXVSTEE,	(long)5L }
{CONF02, IDCONFCL03,	P_MAXVTRANS,	(long)10L }
{CONF02, IDCONFCL03,	P_MAXVSTEE,	(long)10L }
{CONF03, IDCONFCL03,	P_MAXVTRANS,	(long)5L }
{CONF03, IDCONFCL03,	P_MAXVSTEE,	(long)5L }

Tabela 6.6: Exemplo de inicialização de *PCs* indexados.

tanto pela configuração automática, quanto pelo *ED* EDSYS_CYCLE.

Claramente, o uso de parâmetros indexados aumenta a complexidade interna à *API* no acesso aos *PCs* pelos *BFs*. Mesmo com o aumento de complexidade, não deve existir um impacto significativo no desempenho. A manipulação dos índices dos *PCs* pode ser realizada quando os elementos referenciados são alterados, neste caso, o índice discreto é atualizado quando o valor referenciado é alterado e o acesso ao *PC* simplesmente utiliza este índice internamente. Todas as faixas são definidas previamente as execuções e por isso podem ser ordenadas. Esta ordenação garante que a identificação do valor discreto na lista de faixas ordenadas pode ser realizada em ordem logarítmica em função do número de índices discretos. Se o índice for compartilhado por vários *PCs* o impacto é dividido em relação ao número de acessos. No protótipo apenas os *PCsSimples* e *Automáticos* foram implementados.

Funções da *API* para acesso aos parâmetros A *API* de acesso possui funções com tipos específicos, mostradas na Tabela 6.7, para a manipulação dos parâmetros da mesma forma que os *EDs*. As funções são disponíveis tanto para os BFs^U, quanto

Acessam o valor dos *PCs* no ciclo de controle atual		
Função	Descrição	*BF*
*type*_GetPC(PC_{id})	Lê o valor armazenado.	BF^U
*type*_GetPCSync(PC_{id})	Lê o valor armazenado no *PC* no inicio do ciclo, independente se já foi alterado.	BF^U
*type*_SetPC(PC_{id},*value*)	Armazena um valor no *PC*.	BF^U
*type*_SetPCSync (PC_{id},*value*)	Armazena um valor no *PC*, que só estará disponível no próximo ciclo.	BF^U
*type*_GetPCRT(PC_{id})	Lê o valor armazenado no *PC*.	BF^{RT}
*type*_SetPCRT(PC_{id},*value*)	Armazena um valor no *PC*.	BF^{RT}

Tabela 6.7: Funções da *API* utilizadas nos *BFs* para o acesso aos *PCs*.

para os BFs^{RT}, sendo que, a atribuição de valores aos parâmetros manuais pelos BFs^{RT} só podem ser feita de forma instantânea. Isto é uma simplificação para facilitar a implementação com o uso de semáforos, necessários a proteção de sessões críticas internas a plataforma.

6.5 Cálculo da Confiança

Em um controle adaptativo, configurações diferentes são selecionadas com intuito de reagir a alterações do sistema como novos defeitos detectados, alterações da missão ou variações ambientais. A seleção da configuração mais adequada ao estado corrente do sistema é uma questão chave no modelo desenvolvido. A confiabilidade do sistema é um dos indicadores mais importantes e complexos neste processo. Como foi visto na Seção 2, a confiabilidade é a probabilidade do sistema permanecer funcionando corretamente durante toda a duração da missão, ignorando ou não a possibilidade de falhas. Este conceito deve ser transposto para cada configuração possível do controle adaptativo.

A integração entre a percepção e ação do controle é realizada pelo fluxo de processamento, portanto sendo este o objeto chave no cálculo da confiabilidade. A confiabilidade é calculada com base na ocorrência de falhas em função do tempo da missão. Para simplificar o processo de seleção de cada configuração de maneira instantânea, foi definido um índice de confiança, calculado a partir de valores de confiabilidade dos elementos constituintes do sistema. Este índice busca representar de maneira instantânea a probabilidade da execução do fluxo ser um sucesso. Pode-se considerar que o sucesso na execução do fluxo corresponde à execução completa de um ciclo de processamento do fluxo definindo as ações de atuação sem utilizar valores errados provenientes de falhas. Se um valor for identificado com falha, o sistema pode

ser reconfigurado, o que pode ser considerado também como sucesso da execução do fluxo.

> A confiança no fluxo foi considerada como sendo a probabilidade dos valores atribuídos aos *EDs* associados a atuadores serem gerados utilizando valores isentos de falhas. As falhas podem ser tanto de processamento, quanto dos elementos de hardware associados aos *EDs* perceptivos. Se o processamento for interrompido com a detecção de uma falha também é considerado sucesso.

O cálculo da confiança juntamente com o método de diagnóstico são pontos ainda abertos do modelo desenvolvido. Como foi visto na literatura, são muitos fatores que influenciam no sucesso e na confiabilidade de um sistema robótico, muitos destes de difícil modelagem. Nesta seção é apresentada uma abordagem genérica baseada em probabilidades de falhas dos elementos de hardware associados aos *EDs*, na probabilidade de falhas de processamento nos *BFs* e na topologia de conexão do fluxo de dados. Esta abordagem é uma aproximação por não considerar o processamento interno ao código dos *BFs* e por considerar apenas os elementos de hardware associados aos *EDs*. Como pontos a favor, é possível de ser automatizada por um ambiente de desenvolvimento específico e pode oferecer uma estrutura de cálculo de confiabilidade inicial passível de ser refinada pelo projetista.

A probabilidade de um valor no fluxo ser decorrente de uma falha, como foi vista na Seção 6.3.2, é calculado dinamicamente à medida que os *BFs* são executados e os *EDs* acessados. O cálculo é realizado em função da probabilidade dos elementos de hardware (sensores e atuadores) associados aos *EDs* e as probabilidades de falhas de processamento em cada *BF*. Como este valor só fica disponível após cada execução do fluxo, não pode ser utilizado para selecionar uma nova configuração, entretanto, pode ser útil como um indicador para se ativar o processo de adaptação. A seleção de uma nova configuração para o sistema requer que o seu ganho seja calculado sem que o fluxo correspondente seja executado.

As Equações 6.3.1 e 6.3.4 na Página 87 da Seção 6.3.2 calculam a confiabilidade à medida que o valor é selecionado e propagado. Quando existe redundância em atribuições a um *ED* existe um processo de seleção de um único valor, o qual propaga apenas a confiabilidade do selecionado. Para se calcular a confiabilidade sem a execução do fluxo correspondente, deve-se incluir também a probabilidade de seleção de cada caminho no fluxo. A probabilidade de seleção de um valor é a principal diferença entre o cálculo dinâmico e o estático de confiabilidade. Vale ressaltar que os *BFs* de teste associados aos *EDs* podem influenciar diretamente neste cálculo.

A confiabilidade dos *EDs* associados a sensores e atuadores deve ser inicializada com o conhecimento da confiabilidade real do hardware. Esta confiabilidade pode ser alterada diretamente pelo sistema de diagnóstico, ou indiretamente por alterações na

percepção do sistema (*PCs* associados à confiabilidade). O cálculo da confiança de um valor (F_n) produzido por um *BF* é função de suas entradas e da probabilidade de falha deste (r_{F_n}), e é exemplificado na Equação 6.5.1. A confiança de execução de um *BF* e a relação de dependência entre suas entradas e saídas pode ser encontrada na descrição do *BF*.

$$r_{F_n}^{d^j} = r_{F_n} \times \prod_{i \in F_n^{inputs}} r^{d^i} \tag{6.5.1}$$

$$r^{d^k} = \sum_{i}^{red_{d^k}} S_i \times r_{F_i}^{d^k} \tag{6.5.2}$$

$$\sum_{i}^{red_{d^k}} S_i = 1$$

$$S_x = \frac{r_{F_x}^{d^k}}{\sum_{i=1}^{red_{d^k}} r_{F_i}^{d^k}} \tag{6.5.3}$$

$$S_i = f(r_{F_1}^{d^k}, ..., r_{F_{red_{d^k}}}^{d^k}, Testes(d^k)) \tag{6.5.4}$$

A forma genérica do cálculo da confiança de um *ED* no cálculo estático é função da confiabilidade dos *BFs* que produzem o seu valor e de uma probabilidade de seleção S_i mostrada na Equação 6.5.2. Quando não existe redundância o valor de S é igual a 1. Se a seleção é feita somente em função da confiança, a sua probabilidade pode ser a média ponderada da confiabilidade de todos os valores atribuídos (Eq: 6.5.3). Caso exista uma função de teste (Equação 6.5.4), associada ao *ED*, o cálculo fica um pouco mais complexo. Devem-se incluir a probabilidade de detecção de falhas e a probabilidade de se detectar falsas falhas. As probabilidades de S_i, de detecção de falhas, e de detecção de falsas falhas devem ser fornecidas pelo projetista, pois dependem do código utilizado e dos dados analisados. Uma alternativa é utilizar processos de instrumentação, simulação e analise com inserção de falhas para obter valores experimentais, como pode ser visto em mais detalhes na Seção 6.8.

A Tabela 6.8 exemplifica o cálculo de confiança para um *ED*, que recebe dois valores redundantes V_1 e V_2, com as respectivas confianças r_1 e r_2. No *ED* existe um *BF* de teste com probabilidade de detectar falhas reais de t^d e de detectar falsas falhas de t^f. A confiança final do valor do *ED* para o cálculo do restante do fluxo é dada pela Equação 6.5.5.

Falhas	Ocorrência	Falha	Sucesso	
$\varnothing$	$r_1 \times r_2$	$t_1^f \times t_2^f$	$(1-t_1^f) \times (1-t_2^f)$	1
V_1	$(1-r_1) \times r_2$	$(1-t_1^d) \times \frac{r_1}{r_1+r_2}$	$((1-t_1^d) \times \frac{r_2}{r_1+r_2}) + t_1^d \times t_2^f$	2
V_2	$r_1 \times (1-r_2)$	$(1-t_2^d) \times \frac{r_2}{r_1+r_2}$	$((1-t_2^d) \times \frac{r_1}{r_1+r_2}) + t_2^d \times t_1^f$	3
$V_2 \wedge V_2$	$(1-r_1) \times (1-r_2)$	$(1-t_1^d) \times (1-t_2^d)$	$t_1^d \times t_2^d$	4

1	Falha:	Detecta uma falha que não existe nos dois valores.
	Sucesso:	Não detecta falsa falhas nos valores e propaga qualquer um dos dois.
2	Falha:	Não detecta o valor falho de V_1 e o mesmo é propagado.
	Sucesso:	Detecta o valor falho de V_1 ou não detecta e seleciona mesmo assim o V_2.
3	Falha:	Não detecta o valor falho de V_2 e o mesmo é propagado.
	Sucesso:	Detecta o valor falho de V_2 ou não detecta e seleciona mesmo assim o V_1.
4	Falha:	Não detecta o valor falho de V_1 ou de V_2.
	Sucesso:	Detecta o valor falho de V_1 e V_2.

Tabela 6.8: Exemplo da composição da confiabilidade de um *ED* com redundância de dois.

$$
\begin{aligned}
r_d = & (r_1 \times r_2) \times ((1-t_1^f) \times (1-t_2^f)) + \\
& ((1-r_1) \times r_2) \times (((1-t_1^d) \times \frac{r_2}{r_1+r_2}) + t_1^d \times t_2^f) + \\
& (r_1 \times (1-r_2)) \times (((1-t_2^d) \times \frac{r_1}{r_1+r_2}) + t_2^d \times t_1^f) + \\
& ((1-r_1) \times (1-r_2)) \times (t_1^d \times t_2^d)
\end{aligned}
\tag{6.5.5}
$$

O raciocínio utilizado no exemplo anterior com dois valores redundantes pode ser generalizado para níveis de redundância maiores, ou para a aplicação de múltiplos testes de detecção de falhas. É importante destacar que o método mesmo que seja generalizável é uma simplificação do cálculo de uma estimativa da confiabilidade do sistema. Em muitos casos serão necessárias alterações na formulas de cálculo, seja para representar o fluxo interno de dados de um *BF*, ou representar um critério mais elaborado de seleção do valor redundante embutido no teste do *ED*. Por exemplo, pode-se utilizar nos testes, mecanismos de votação ou seleção de valores por proximidade.

Estruturas de dependência mais refinadas e processos de seleção mais elaborados podem ser incluídos futuramente na descrição dos *BFs*, caso seja utilizado um ambiente de desenvolvimento específico. No protótipo desenvolvido existe um *BF* que multiplexa um conjunto de entradas em um conjunto de saídas. Cada saída depende simultaneamente apenas de dois *EDs*, um que controla a multiplexação e o outro

ED com o valor que deve ser propagado. Para simplificar o cálculo de confiança, a associação entre as entradas e saídas foi considerada aleatória. Desta forma, cada entrada contribuí proporcionalmente com a confiança de todas as saídas.

Outro ponto importante de se destacar é o uso no modelo de *EDs* de memória, que armazenam explicitamente no fluxo um valor gerado para um *ED* de um ciclo para outro. Neste caso, existem duas abordagens para o cálculo da confiança: na primeira que é mais simples, o valor de confiança do *ED* é também copiado junto com o valor do dado de um ciclo para outro; na segunda opção é associado ao *ED* todo o cálculo de confiança utilizado na sua formação. A segunda opção além de muito mais complexa, só tem sentido se houver alterações drásticas de confiabilidade de um ciclo para outro que devem ser analisadas pelo controle adaptativo. Vale ressaltar, que alterações drásticas de confiabilidade provavelmente devem ser tratadas na recuperação de falhas e não no controle adaptativo.

6.5.1 Cálculo do índice de confiança

Após definir o processo de cálculo ao longo do fluxo restam duas questões. Como criar um índice único para todo o fluxo capaz de representar a sua probabilidade de falhas adequadamente. Como variar a importância da confiança de determinados elementos do sistema em função da missão ou fase corrente. Estes dois pontos foram resolvidos no modelo de forma integrada.

Primeiramente se imaginou que o sucesso do fluxo é gerar valores corretos para a atuação. Assim sendo, se for criado um índice compondo a confiança de todos os *EDs* associados a atuadores poderia se obter um valor representativo único. Entretanto, a importância do atuador pode variar muito para o sistema dependendo da sua função e da tarefa sendo realizada. Por exemplo, um farol como atuador pode ter uma importância muita pequena durante o dia e muito grande durante a noite. Podem existir pontos no sistema cuja confiança seja muito significativa ou totalmente irrelevante para o sucesso da missão ou da fase.

A solução encontrada foi permitir a definição de um fator de relevância de confiança para cada *ED* do fluxo, sendo este definido pelo projetista e associado através da fase da missão. O índice é composto então em duas etapas distintas:

1. O cálculo de confiança de todos os *EDs* que podem ser utilizados para um determinado fluxo. Este cálculo é realizado a partir dos *EDs* que tem confiança determinada inicialmente ou pelo sistema de diagnóstico, sendo propagado pelos *BFs* do fluxo, inclusive os de testes.

2. A confiança dos *EDs* relevantes é combinada por uma soma ponderada (Eq.: 6.5.6). A soma pode ser substituída se desejado por outras funções mais elaboradas.

$$I^C = \frac{\sum_{i \in EDs} I_i \times r_i}{\sum_{i \in EDs} I_i} \tag{6.5.6}$$

O vetor de relevância de cada *ED* ($I_i | i \in EDs$) pode ser definido inicialmente igual para todos os atuadores. O projetista pode alterá-lo na definição da fase de forma a personalizar o cálculo. É possível inclusive tornar relevantes *EDs* associados somente a testes ou a processos de diagnóstico.

6.5.2 Processamento do índice de confiança

O processo completo deste cálculo se apresenta à primeira vista complexo e caro em termos de processamento. Entretanto, existem alguns pontos importantes que facilitam a sua implementação e reduzem o impacto no processamento total do sistema. No protótipo implementado este cálculo foi feito utilizando funções criadas manualmente.

- Todo o cálculo é iniciado a partir da confiabilidade dos *EDs* associados ao hardware. Neste caso, apenas atualizações provenientes do sistema de diagnóstico ou de eventos específicos associados à confiabilidade promovem alterações no resultado do cálculo. Assim sendo, estes eventos são fáceis de serem supervisionados de forma a evitar recálculos desnecessários.

- O cálculo de confiança de um fluxo se baseia na sua composição de *BFs* a qual não varia após a sua definição. Neste caso, uma função na linguagem *C* específica que realiza o cálculo pode ser gerada automaticamente de forma a otimizar o processamento.

- O cálculo se baseia no fluxo, que em grande parte é compartilhado com várias configurações diferentes, principalmente se corresponderem a adaptações da mesma fase. Neste caso, vão existir sub-cálculos iguais associados aos subfluxos iguais. Portanto, o cálculo pode ser particionado em blocos de forma a gerar resultados intermediários comuns a várias configurações diferentes. Estas partições podem ser geradas previamente a execução. O custo adicional desta abordagem de cálculo particionado é essencialmente o armazenamento dos resultados intermediários necessários.

- Este cálculo é utilizado no processo de adaptação e não no processo de recuperação de novas falhas que necessita ser rápido. Assim sendo, pode ser realizado ao longo de vários ciclos de processamento do fluxo, de forma a não causar impactos negativos processamento do sistema. A divisão e o compartilhamento de cálculos de subfluxos facilita muito este aspecto.

6.6 Diagnóstico de defeitos

Neste trabalho não foi proposta nenhuma solução nova para o problema de realização de diagnósticos. Nesta seção são destacados os pontos fundamentais e requisitos de integração das abordagens de sistemas de diagnóstico com o modelo proposto. Foi considerado que o processamento do diagnóstico também é realizado por *BFs* especiais que participam também do fluxo, como será visto na Seção 8.2. Nos processos de tempo real os recursos de detecção e recuperação de falhas são implementados com restrições extremamente rígidas de tempo. O processo de diagnóstico nem sempre pode atender limites de tempo extremamente rígidos, sendo realizado em um tempo maior e mais variável.

O diagnóstico é muito importante ou fundamental para a implementação da tolerância a falhas de maneira geral por vários motivos, entre eles: a seleção da ação de recuperação adequada a uma determinada falha detectada, pode ser dependente do diagnóstico exato do defeito; se a falha pode ser originada de vários módulos diferentes e o defeito não for assinalado a um elemento específico, todos os módulos sob suspeita devem ser isolados. Esta atitude pode representar uma grande perda nos recursos disponíveis de um sistema provocando uma perda de desempenho ou uma perda dos serviços ou tarefas oferecidas.

A estrutura de conexão entre os *BFs* e os *EDs* define diretamente as dependências de informações existentes no fluxo de processamento. Falhas detectadas neste fluxo podem ser propagadas do ponto de detecção até suas entradas correspondentes, identificando assim os *EDs* e *BFs* que podem ter gerado as informações incoerentes. Neste caso, como alguns *EDs* estão associados com hardware pode-se obter um diagnóstico de alguns defeitos. O espaço de observação é definido pelos *EDs* associados a elementos de hardware e a indicadores internos.

A estrutura de dependências de informação existente no fluxo do modelo é um bom inicio para o diagnóstico, mas na maioria dos casos, pode não conter informações suficientes. O fluxo de processamento não contém informações fundamentais de interdependências entre atuadores, elementos mecânicos ou estruturais, ou hardware e software utilizado no processamento de sinais. Estas informações podem ser essenciais para realizar diagnósticos satisfatórios.

Utilizando como exemplo o esquema do *Nomad* mostrado na Figura 9.2 na Página 186, considere que é associado no modelo um *ED* para cada sensor distinto. Quando um valor fora do domínio esperado é percebido em um *ED* indicando a presença de uma falha, o defeito pode estar no próprio sensor associado, ou na placa *Intellisys 100* ou conexão entre eles. Caso o defeito não possa ser isolado apropriadamente, é necessário que a confiabilidade de todos os elementos envolvidos seja reduzida, influenciando indiretamente na confiabilidade dos demais sensores. Este tipo de informação nunca estará presente um fluxo de processamento. Neste caso,

para realizar um diagnóstico mais refinado é necessário o uso de estruturas de dependências que complemente a informação já existente na topologia do fluxo.

A detecção de falhas no modelo é realizada internamente aos *BFs* que retornam a *PCA* a ocorrência de uma falha, juntamente com informações que possam ser relevantes para o diagnóstico. Por exemplo, um *BF* de teste associado a um *ED* pode retornar um evento de falha que identifique qual ou quais *BFs* podem ter gerado a informação incoerente. Eventos de sucesso são gerados da mesma forma quando um teste não detecta falhas. O sistema de diagnóstico deve receber estes eventos, que ficam associados a *EDs* e *BFs*, e processa-los de forma a identificar a origem do problema.

Quando é detectada uma falha o sistema de diagnóstico deve aumentar a probabilidade de falhas dos *EDs* e *BFs* suspeitos, reduzindo a confiabilidade destes e consequentemente das adaptações que dependem de suas informações (Seção 6.5). Desta forma o sistema de diagnóstico interage com o controle adaptativo.

O controle adaptativo considera para suas decisões apenas o índice de confiança calculado a partir da confiabilidade dos *EDs* e *BFs*, mesmo que o espaço de falhas e de diagnóstico seja muito maior. Esta é uma simplificação aceitável, pois se for diagnosticado um defeito em um elemento de hardware mesmo que indiretamente associado a *EDs* do fluxo este deverá sofre alterações em sua confiabilidade influenciando no controle adaptativo. A viabilidade da execução de uma configuração do controle vai depender diretamente da confiança dos *EDs* e *BFs* utilizados influenciada pela confiabilidade de elementos externos ao fluxo associados por estruturas de dependências e atualizados pelo sistema de diagnóstico.

Existem várias linhas de pesquisa sobre diagnóstico, e praticamente todas utilizam informações de interdependência entre os elementos constituintes do sistema. No modelo proposto, não foi necessário fixar uma metodologia de diagnóstico específica, mas qualquer uma selecionada deve atender um conjunto de requisitos:

- Os *BFs* e *EDs* devem ser mapeados nos elementos diagnosticáveis do sistema, Ou seja, o sistema de diagnóstico utilizado deve ser capaz de identificar falhas nos elementos do modelo de controle.

- O sistema de diagnóstico deve funcionar recebendo os eventos relativos aos resultados de testes associados aos *BFs* e *EDs* efetuados no fluxo de processamento.

- O sistema de diagnóstico deve considerar, juntamente com os eventos de teste, a topologia do fluxo específica da adaptação na qual o teste foi executado. Considerando associado a cada evento de teste um subconjunto das dependências existente na configuração específica.

- O sistema de diagnóstico deve atualizar adequadamente a confiabilidade dos *BFs* e *EDs* do modelo.

- É desejável que algumas funções de processamento dos *BFs* sejam capazes de produzir diretamente eventos de teste ou outras informações que influenciem na confiabilidade de elementos internos ou externos ao fluxo. Este requisito visa permitir o projetista implementar métodos muito específicos ou mais elaborados de analise de defeitos capazes de interagir com o sistema de diagnóstico.

- É desejável que o processamento do sistema de diagnóstico possa ser efetuado progressivamente em vários ciclos do controle. De forma a não impactar significativamente no desempenho do sistema. Além disso, deve ser capaz de suportar o acumulo de diversos eventos de teste antes de serem processados. Estes requisitos visam facilitar a gerencia dos recursos de processamento do sistema, quando este não se encontra em situações de falhas críticas. Vale lembrar que em momentos de detecção de novas falhas ou em estados de recuperação, as tarefas mais importantes para o sistema podem ser os testes e o processamento do diagnóstico.

- Deve ser possível introduzir no sistema de diagnóstico, as probabilidades de falhas iniciais nos elementos mapeados e outras informações históricas apropriadas.

- As adaptações, fases e missões podem ser inibidos por baixos valores de confiança do fluxo (Seção 6.5.1). O sistema de diagnóstico pode associar diretamente a cada adaptação, fase ou missão um subconjunto de elementos cujo funcionamento correto é essencial para o sucesso. Neste caso, o sistema de diagnóstico pode interagir diretamente com o controle de alto nível e inibir completamente opções não confiáveis de forma simples e eficiente.

O sistema de diagnóstico deve ser atualizado quando existem novas evidências de sucesso ou falha que podem alterar a confiabilidade dos elementos. O sucesso pode significar a recuperação de um elemento de hardware depois de uma falha instantânea ou transiente. Neste caso, pode ser necessária uma confiabilidade maior para reintegrá-lo ao processamento que a confiabilidade utilizada para considerá-lo com defeito. Ou seja, mais eventos de sucesso para reintegrá-lo, do que eventos de falha para considerá-lo defeituoso.

6.6.1 Diagnóstico de atuadores

Um dos recursos de diagnóstico mais difícil de se implementar é a utilização de modelos funcionais, principalmente quando existe interação direta com o meio. A correlação entre as ações executadas e os efeitos futuros esperados é sempre de difícil generalização, no caso dos robôs, e de difícil inserção em um sistema de diagnóstico

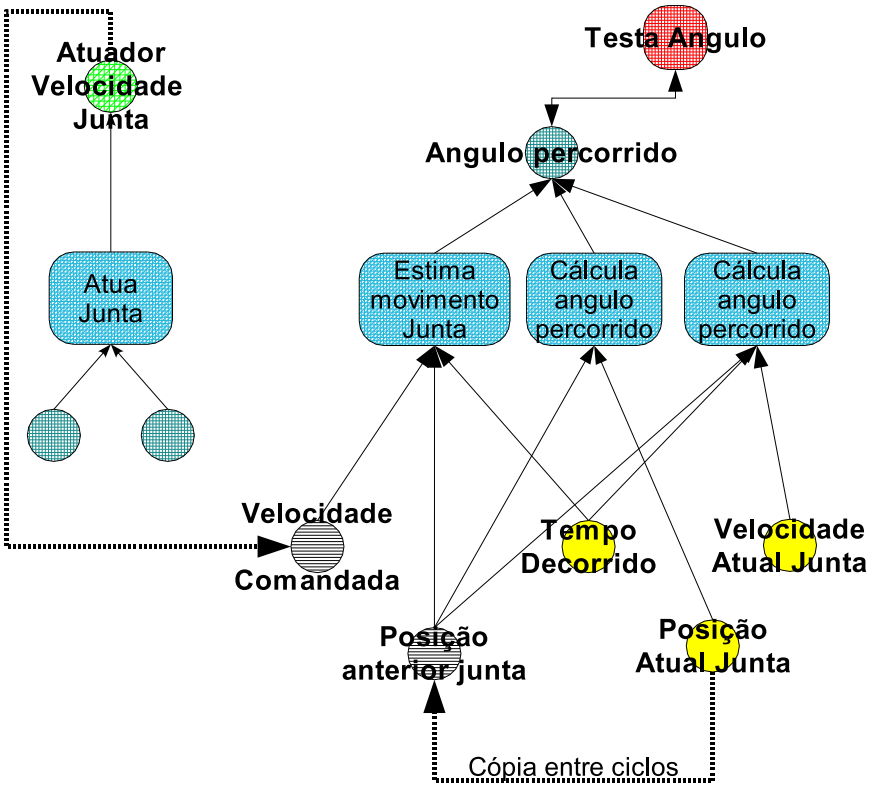


Figura 6.10: Inserção de testes para atuadores no fluxo de processamento.

genérico. Quando o modelo foi desenvolvido, ficou clara a necessidade de permitir a inserção deste tipo de recurso diretamente no fluxo. Um exemplo desta inserção é mostrado na Figura 6.10.

Com o uso de *BFs* normais e *EDs* de memória, pode ser possível tratar falhas de atuação da mesma forma que são detectadas falhas sensoriais. Os *EDs* de memória como visto na Seção 6.3 copiam informações de ciclo a outro, neste caso, o *BF* de teste associado ao *EDÂngulo Percorrido* pode indicar falhas tanto nos sensores que coletaram dados no ciclo atual, quanto no atuador que recebeu um comando no ciclo anterior. O diagnóstico do atuador pode ser feito da forma equivalente a falhas nos sensores, isto se forem associados corretamente aos *EDs* de memória as dependências de hardware existentes. No exemplo mostrado, o *ED* de memória *Velocidade Comandada* deve ser associado ao hardware do atuador, e o *EDPosição Anterior Junta* ao hardware do sensor associado ao *EDPosição Atual Junta*.

Um outro ponto importante é lembrar que tanto o teste como o *BF* que faz a estimativa de funcionamento da junta podem ser ativados e desativados de acordo com os requisitos de confiabilidade no atuador.

Além da possibilidade de se inserir outros tipo de teste no fluxo, é possível criar

ciclos de configurações com características interessantes. Um recurso muito interessante com o uso da arquitetura proposta é a possibilidade de inserir fases com mais redundância e testes, intercaladas com fases com pouca redundância e um desempenho melhor. Além disso, é possível também intercalar diversas fases distintas e equivalentes nas quais a redundância utilizada e os testes efetuados variam. Estas possibilidades permitem a combinação de fases com maior desempenho com fases com maior ganho de conhecimento sobre o estado do sistema, maximizando assim a probabilidade de sucesso nas missões.

Se o sistema de diagnóstico reduzir gradualmente a confiabilidade de elementos de hardware os quais não estão sendo testados nas configurações atuais, a confiabilidade geral deve cair e a importância de efetuar testes deve crescer progressivamente. Em um dado momento, o controle adaptativo deve selecionar uma configuração que proporcione um ganho maior de informação sobre o estado do sistema. Se o controle adaptativo e o sistema de diagnóstico estiverem bem ajustados, o sistema sempre vai balancear tanto o desempenho quanto o autoconhecimento.

6.7 Rede Baysiana de diagnóstico

O objetivo do trabalho desenvolvido não foi criar novos métodos de diagnóstico. Mas criar um modelo de controle estruturado e coerente que permita para o diagnóstico, o uso dos métodos mais simples aos mais elaborados.

Uma sugestão interessante para o diagnóstico é o uso de redes Baysianas [Pearl, 1988] por atender os requisitos apresentados. Um dos maiores problemas no uso destas redes para o diagnóstico é obter as informações de dependência adequadas para a construção da mesma ([Nikovski, 2000]). Como já citado, a topologia de dependências existente no modelo pode ser utilizada para iniciar a construção da rede, principalmente por já possuir algumas das probabilidades necessárias para a implementação do processo de diagnóstico.

Os nodos da rede podem representar diretamente os *BFs* e *EDs* e as probabilidades de falhas podem ser obtidas de dados coletados ou de informações provenientes dos fabricantes dos componentes.

A rede deve processar os eventos de falhas e sucesso gerados pelos *BFs* que realizam testes. Como a estrutura de dependência geral é estática e as adaptações são calculadas previamente a execução do sistema, é possível também calcular previamente todas as formulas de propagação dos eventos e atualização de probabilidades na rede. Reservando memória para armazenar as formulas de propagação é possível implementar a rede de forma eficiente.

6.8 Modos de execução

No modelo criado, a Plataforma de Controle Adaptativo (*PCA*) controla a execução e comunicação do fluxo. Esta característica oferece três possibilidades interessantes para facilitar o desenvolvimento de um projeto.

1. A plataforma é responsável pela execução do fluxo de processamento dos *BFs*. Neste caso é possível medir o tempo de execução de cada fluxo específico e de cada *BF* individualmente, obtendo dados de custo de processamento para cada configuração. Estes dados podem ser utilizados no controle adaptativo.

2. A *PCA* controla a comunicação, assim sendo, existe a possibilidade de acessar os valores de cada *ED* gerado no processamento do fluxo. Estes valores podem ser utilizados para ajuste ou calibração de funções próprias do robô ou para detecção de falhas nas comparações efetuadas nos *EDs*.

3. A estrutura criada no modelo composto por *BFs* e por *EDs* oferece um bom isolamento entre os elementos de processamento. Este isolamento permite um alto grau de independência na execução de cada *BFs* individualmente, assim sendo, suas funcionalidades podem ser facilmente exercitadas em um ambiente controlado.

A tolerância a falhas adaptativa se baseia na otimização do uso dos recursos disponíveis em um sistema da melhor forma possível, se adaptando dinamicamente a mudanças na situação corrente. Para tanto, os controle adaptativo deve ser capaz de avaliar as suas alternativas de configuração, o que envolve o conhecimento dos custos e ganhos associados a cada uma. O controle deve possuir previamente dados que permitam calcular os fatores de custo e desempenho necessários, e se possível, atualizar estes dados dinamicamente, garantindo assim, que os cálculos sempre correspondam o melhor possível à realidade do sistema.

Sistemas que interagem com o ambiente real, como já foi citado várias vezes, necessitam parâmetros individuais para calibrar determinados elementos como sensores ou atuadores e para permitir a detecção de falhas. No caso de detecção de falhas, os parâmetros devem ser coletados no próprio sistema, quando este está em funcionamento considerado normal sem a presença de falhas. Durante o funcionamento considerado normal de um sistema, as diferenças observadas nos valores de um mesmo *ED*, no qual existe redundância, podem ser considerados como desvios ou incertezas normais devido às próprias características inerentes ao robô. Para se realizar estas comparações, análises estatísticas ou até mesmo correlações de dados, é necessário possuir os dados gerados pelo fluxo. O projetista pode inclusive testar outros métodos de detecção de falhas através de ferramentas matemáticas externas.

Qualquer sistema pode ser alterado para exportar dados internos, mas a facilidade oferecida pela estrutura dos *EDs* e *PCA* pode proporcionar, além de uma padronização do formato de exportação, a possibilidade de ativar a exportação através de atributos associados aos *EDs*. Desta forma, fica mais fácil à integração com ferramentas externas de análise.

Infelizmente, a coleta de dados de um sistema, principalmente de tempo real, pode interferir no seu funcionamento esperado. Considerando a necessidade e possibilidade de acesso a dados úteis através da plataforma de controle, foi definido um conjunto de modos diferentes de execução desta. Estes modos objetivam facilitar a coleta seletiva de dados e indicadores e um ajuste gradual do sistema. Estes modos de operação da *PCA* são aderentes ao processo de desenvolvimento, e podem ser controladas por atributos dos *EDs* e *BFs* e por comandos específicos. No protótipo desenvolvido foram implementadas apenas poucas opções, estas controladas por parâmetros de compilação.

1. **Teste intensivo**- A execução de cada *BF* do fluxo depende exclusivamente dos seus *EDs* de entrada e de seu estado interno acessível e salvo pela *PCA*. Portanto é possível realizar uma etapa em todos os todos os *BFs* são executados intensivamente. Os valores de entrada podem ser gerados automaticamente de forma aleatória, ou não, dentro do domínio válido de cada *ED* de entrada. Este tipo de teste possui dois objetivos principais: obter uma estimativa inicial do tempo de processamento e variações na execução de cada *BF*; exercitar o código interno a cada *BF* de forma isolada e controlada focando principalmente condições de contorno nas entradas. Vale ressaltar, que este tipo de teste detecta somente erros simples e exceções que poderiam ocorrer durante a execução do sistema completo, sem analisar diretamente ou significativamente a funcionalidade e confiabilidade do código programado. Além disso, apenas em casos muitos simples é possível testar todas as opções existentes. Mas como o objetivo é o desenvolvimento de um sistema robusto, este tipo de simulação continua válido.

2. **Teste inicial** - Nesta etapa, todos os fluxos de *BFs* distintos são executados na plataforma de destino, inclusive incluindo as tarefas de tempo real rígido (BFs^{RT}). O objetivo desta etapa é aferir os tempos de processamento de cada topologia possível de *BFs* que possa vir a ser utilizada no controle. Os tempos associados ao processamento de testes associados aos *EDs* e *BFs* e nas funções de diagnóstico também são estimados. Os valores de *EDs* associados à atuação podem ser fixados em padrões de forma a permitir um teste inicial seguro. Os valores produzidos pelo fluxo de processamento e pelos sensores podem ser exportados, permitindo uma analise externa ao controle.

3. **Aprendizado dos parâmetros de teste** - O controle real contendo as várias configurações é executado. O objetivo desta é exercitar todos os estados de controle possíveis e todas a formas de redundância existente no controle. São coletados os valores atribuídos a todos os *EDs*, que podem apresentar redundância ou sejam pontos potenciais de teste. A coleta destes valores considera que o robô ou o sistema em questão esta em perfeito funcionamento. As funções de detecção de falhas associadas aos *EDs* funcionam em um modo de coleta de dados e não de detecção de falhas. Nesta opção também são coletados valores de desempenho na execução de tarefas associados as possíveis reconfigurações do sistema. Se existem na missão fases equivalentes, todos os caminhos excluindo as transições de equivalência (Seção 7.1.3) são executados para coleta de tempo.

4. **Processamento do Histórico** - A coleta de dados internos ao fluxo pelos *EDs* permite a criação de histórico ou *log* de execução. Este pode ser processado com intuito de extrair os parâmetros que serão utilizados na detecção das falhas e no diagnóstico. Os limites máximos entre valores equivalentes gerados e atribuídos aos mesmos *EDs* são estimados. Os parâmetros de custo e tempo de processamento de cada topologia também são aferidos. Esta analise pode incluir a formação de agrupamentos e correlações de dados para relacionar valores de detecção de falhas com as possíveis configurações do sistema ou com valores de outros *EDs*. Em outras palavras, em vez de definir limites únicos para detecção de falhas para todos os estados do sistema, podem-se gerar limites associados a estados específicos.

5. **Inserção automática de falhas** - O controle é executado em um modo de simulação e depuração utilizando o histórico de dados e sensores coletados. Em uma primeira etapa as execuções anteriores são simuladas através da inserção automática dos valores coletados nos *EDs* associados aos sensores e controles internos. As funções de detecção de falhas são ativadas no modo normal. Em uma segunda etapa, o teste é repetido com a inserção automática de erros nos valores associados aos *EDs*, de forma a exercitar e depurar os processos de detecção, isolamento e recuperação das falhas. Esta etapa também é útil para o ajuste das probabilidades de detecção de falhas.

6. **Redução do Grafo de Controle Adaptativo** - Após estas várias etapas é possível selecionar um conjunto de configurações que apresente ganhos significativos para o robô. Por exemplo, se existirem duas configurações com diferença de desempenho insignificante e a redundância de uma é subconjunto da outra, pode-se remover a configuração de menor redundância e confiabilidade. Com estes critérios o *GCA* final pode ser gerado.

7. **Funcionamento Normal** - O controle é executado utilizando o grafo gerado anteriormente e os parâmetros de detecção de falhas atribuídos adequadamente. Todas as funções de tolerância a falhas são ativadas. Nesta etapa o controle esta pronto. É possível se inserir falhas por software no fluxo através dos para validar a recuperação de falhas e o próprio controle adaptativo.

Estas etapas compõem um ciclo básico de projeto proposto para o modelo. Não são rígidas, podendo ser repetidas várias vezes de forma interativa, objetivando sempre a otimização do controle e da tolerância a falhas.

É claro que muitos destes ajustes poderiam ser feitos dinamicamente pelo próprio controle, ao longo de sua utilização. Já existem trabalhos mais recentes com estas abordagens, como o da Paker [Parker, 1999]. Neste trabalho deixamos esta questão como possibilidade para trabalhos futuros 10.1.

Capítulo 7

Controle Híbrido

> As vitórias em batalha não poderão jamais, serem repetidas - as circunstâncias de cada combate são mutáveis e exigem uma resposta própria e particular.
>
> Sun Tzu

A arquitetura de controle escolhida para o modelo, como já foi citado muitas vezes anteriormente é a híbrida. Entende-se por híbrido, um controle dividido em diferentes níveis com abordagens distintas. O nível mais alto de controle utiliza um de um autômato ou máquina de estados finitos com estados discretos que definem seqüências elaboradas de ações ou situações. Cada estado discreto habilita em um nível mais baixo do controle processos que efetivamente processam a percepção e ação do robô. O nível mais baixo é implementado pelo fluxo descrito no capítulo anterior.

As arquiteturas com abordagens híbridas buscam unir a simplicidade das máquinas de estados finitas, capazes de definir seqüências complexas, com processamento da percepção e das ações realizado de forma simples e eficiente. A arquitetura básica pode ser vista na Figura 3.4 na Página 31. Esta arquitetura de controle foi selecionada o para o modelo devido a um conjunto de razões, algumas das quais destacamos a seguir.

- Quando se investe no desenvolvimento de um sistema para que este possua características de tolerância a falhas, provavelmente o mesmo terá que realizar tarefas críticas ou ficar inacessível por um longo período de tempo. Portanto, o seu controle deve ser capaz de realizar tarefas complexas ou missões muito extensas, nas quais o seu uso varia ao longo do tempo. O modelo de controle utilizado, portanto deve permitir a descrição de seqüências complexas de ações

ou tarefas, permitindo compor e alterar missões de acordo com o objetivo atual ou estado do sistema. Utilizando máquinas de estado ou autômatos finitos é fácil compor seqüências complexas.

- O sistema sempre deve tentar maximizar a chance de sucesso na execução da sua missão, mesmo na presença de defeitos. Deve ser possível ao projetista definir seqüências de ações ou estados alternativos para a realização de uma mesma tarefa, de preferência utilizando elementos de hardware sistema diferentes. Neste caso, quando é detectada uma falha que impeça a execução da fase corrente, o sistema pode selecionar se existir uma fase alternativa equivalente que não dependa dos elementos defeituosos. Em um mesmo autômato é possível coexistirem muitas seqüências distintas de estados conectados ao mesmo estado inicial e ao um mesmo final, o que facilitar o uso de estratégias diferentes para se alcançar o mesmo objetivo.

- O sistema deve ser capaz de realizar missões distintas, definidas em alto nível de forma a maximizar sua utilidade mesmo na presença de falhas. Se o robô é capaz de realizar missões distintas que dependem de hardware diferentes, é possível que ele troque de missão na presença de defeitos. Esta capacidade de realizar missões diferentes é importante para a tolerância a falhas em times cooperativos, pois pode permitir a redistribuição das tarefas entre os integrantes do time, quando um membro não esta realizando suas funções adequadamente. O modelo de controle deve ser capaz de descrever várias missões, além de identificar as suas dependências em função dos recursos disponíveis. A possibilidade de agrupar vários autômatos distintos em um maior através da inserção de estados intermediários e novas transições facilita a união de missões distintas em uma única máquina de estados para o controle.

- Quando se desenvolve um projeto prevendo a existência de defeitos, como é o caso dos sistemas tolerantes a falhas, é necessário definir a ação a ser realizada, no caso de se detectar a presença de falhas ou detectar uma situação inesperada. O ideal para um modelo de controle é facilitar a inclusão destas ações alternativas. Além disso, deve ser possível identificar todas as possíveis seqüências de ações que acontecem quando o sistema entra em um estado não previsto, para prevenir danos, ou promover uma recuperação mais adequada. Os objetivos destes estados alternativos são conduzir o sistema a um estado seguro, recuperar de falhas detectadas ou até obter mais informações sobre o sistema para se selecionar a ação mais adequada. A inclusão destas alternativas pode ser feita facilmente utilizando-se os autômatos finitos.

MISSÃO$_{id}$	Descrição da missão
$M\_GOTOGOAL$,	Percorre o ambiente até uma posição XY evitando obstáculos.
$M\_MAPPING$,	Percorre o ambiente criando um mapa de obstáculos.
M_DIAG ,	Aguarda comandos externos de diagnóstico ou reparo.

Tabela 7.1: Exemplo de missões.

7.1 Autômato de controle

Os autômatos finitos determinísticos, também conhecidos como máquinas de estados finitos, é uma estrutura abstrata amplamente utilizada no desenvolvimento de algoritmos em muitas áreas distintas do conhecimento. Uma máquina de estados finitos ou um autômato finito é uma quíntupla $M = (Q, \Sigma, \delta, q_0, F)$. Os elementos constituintes são os seguintes:

Q – Conjunto finito de estados do autômato.

Σ – Conjunto finito de símbolos que ativam as transições. Correspondem no modelo as condições que ativam as transições de estados.

$q_0 \in Q$ – Estado inicial do autômato.

$F \in Q$ – Estados finais do autômato ou estados de aceitação.

δ – A função de transição determinística do autômato. Corresponde a uma função de $Q \times \Sigma$ para Q.

$$f(Q, \Sigma) \rightarrow Q$$

O nível mais alto do controle do sistema é definido por uma ou mais missões distintas. Cada missão é representada por um autômato próprio M^m. Os estados do autômato Q^m correspondem às fases distintas da missão. O robô é capaz, portanto de realizar um conjunto de missões, sendo cada uma definida por um autômato composto por fases específicas. As transições entre as fases (δ^m) são ativadas por condições simples (Σ^m) realizadas sobre os valores gerados pelo fluxo de processamento no nível mais baixo do controle. A seleção da missão corrente é proveniente de um comando externo ao modelo, permitindo a sua inclusão em times de robôs com mecanismos próprios de distribuição de tarefas, ou um uso de um planejador adequado.

Um exemplo de hierarquia básica é mostrado na Figura 7.1 e na Tabela 7.1. Para cada missão é definido um conjunto de fases associadas como exemplificado na Tabela 7.2.

MISSÃO$_{id}$	$Fase_{id}$	Descrição da fase
M_SYSTEM,	$F\_MISSION$,	"Parado aguardando uma nova missão."
M_SYSTEM,	$F\_FAILSTOP$,	"Parado aguardando reparos."
$M\_GOTOGOAL$,	$F\_STOPPING$,	"Para o robô."
$M\_GOTOGOAL$,	F_GOAL,	"Chegou ao objetivo."
$M\_GOTOGOAL$,	$F\_GOTOGOAL$,	"Movimenta diretamente para o objetivo."
$M\_GOTOGOAL$,	$F\_ALIGNSLINE$,	"Alinha com o objetivo."
$M\_GOTOGOAL$,	$F\_ALIGNWALL$,	"Alinha com um obstáculo a direita."
$M\_GOTOGOAL$,	$F\_GETAWAY$,	"Acompanha o obstáculo a direita."
$M\_GOTOGOAL$,	...,	"..."

Tabela 7.2: Exemplo da hierarquia de controle - Missão X Fases.

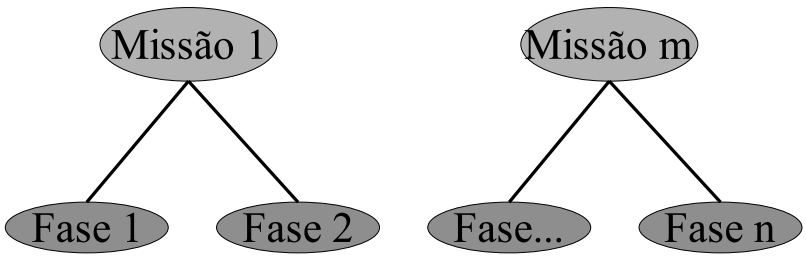


Figura 7.1: Hierarquia básica.

Atributo	Sub-atributo
Transições de Fase	Testes de Transição
Fases equivalentes	Índice de Sucesso
Fases de Recuperação	
Configuração do controle adaptativo	Índice de confiança
	Índice de desempenho
	Função de ganho
Controle do fluxo de baixo nível	Configurações de *PCs* ativas
	Conjunto dos *BFs* ativos

Tabela 7.3: Conjunto de atributos de definição de uma missão.

MISSION:	M_GOTOGOAL
MIN_IPERF:	0.30
MIN_IREL:	0.70
MIN_IPROFIT:	0.90
PHASE:	F_GOTOGOAL
PC_IDCONF:	CONF02
IPERF(I_D):	CYCLE_PCVEL
MIN_IPERF:	0.50
IREL(I_C):	STDCALC
MIN_IREL:	0.80
IPROFIT(I_G):	STDLINEAR
MIN_IPROFIT:	1.00
ESSENTIAL_BFS:	GoToGoal
...	...

Tabela 7.4: Exemplo de dados de descrição de uma missão.

7.1.1 Definição das fases de missões

A estrutura básica do controle de alto nível é a definição das fases e de seus atributos. Os atributos controlam a funcionalidade do controle de baixo nível, além de determinar requisitos ou objetivos instantâneos. Os atributos definidos para as fases do modelo são destacados na Tabela 7.3 e exemplificados na Tabela 7.4, sendo descritos com mais detalhes ao longo da seção.

7.1.2 Transições entre as fases de missões

No controle de um sistema complexo como um robô, muitos critérios podem ser utilizados no alto nível para definir as condições ou eventos que ativam as transições

Fase de Origem	Fase de Destino	$Teste_{ID}$
GetAway	GetBack	limitIdist
GetAway	Goal	sucess&!colision
GetAway	stopping	colision

Tabela 7.5: Exemplo de dados de uma fase do modelo.

de fase. Dentro destas condições podemos destacar valores gerados pela percepção, indicadores internos do sistema ou até um contador de tempo ou ciclos. No modelo optou-se por concentrar o processamento sempre que possível no nível mais baixo do controle implementado pelo fluxo de processamento. A solução decorrente desta opção é basear todas as transições de fase definidas pelo projetista em valores associados aos *EDs* e aos *PCs*. Quando for necessário algum processamento complexo para a decisão de uma transição, este processamento deve ser realizado por um ou mais *BFs* e atribuído a um *ED*.

As condições para as transições ou Testes de Transição foram definidas como operações e comparações simples realizada sobre um conjunto de fatores. Duas formas foram definidas para um teste, visando simplificar a definição de condições pelo projetista. A primeira forma é uma comparação simples entre dois fatores, exemplificada na Tabela 7.6. Na segunda forma é incluída uma operação matemática simples aplicada com um terceiro fator. Esta forma é mostrada na Tabela 7.7. Os componentes dos testes são os seguintes:

$Teste_{ID}$ – Identificador único da condição específica. Este identificador é textual para facilitar o uso na definição das transições.

$Fator_{ID}$ – Identificador do operando utilizado no teste. Este fator pode corresponder aos seguintes tipos:

$V(ED_i)$ O valor armazenado no *ED* índice i.

$R(ED_i)$ O valor da confiabilidade associada ao *ED* índice i.

$P(PC_i)$ O valor armazenado no parâmetro de controle *PC* índice i.

$Const$ Um valor constante.

$Comp_{ID}$ – Identificador textual da comparação realizada no teste. A lista implementada é mostrada na Tabela 7.8. É importante ressaltar que é simples incluir novas comparações.

$Oper_{ID}$ – Identificador textual da operação matemática realizada entre os dois primeiros fatores do teste. A lista implementada é mostrada na Tabela 7.9.

$Teste_{ID}$	$Fator_{ID}$	$Comp_{ID}$	$Fator_{ID}$
lowbat	VLOWBAT	NEQ	ZERO
			VLOWBAT$\neq 0$
colision	VCOLISION	NEQ	ZERO
			VCOLISION$\neq 0$
sucess	VDISTGOAL	LST	PMAXDISTERR
			VDISTGOAL<PMAXDISTERR
limitIdist	VDISTPI	GRT	PDISTPI
			VDISTPI>PDISTPI
alignedGoal	VGOALANG	ALST	PALIGNANG
			ABS(VGOALANG)<PALIGNANG

Tabela 7.6: Exemplo de comparações para ativar as transições.

$Teste\_ID$	$Fator\_ID$	$Oper\_ID$	$Fator\_ID$	$Comp\_ID$	$Fator\_ID$
$FarIdist$	$VDISTPI$	SUB	$PDISTPI$	GRT	$PDISTPI$
					$(VDISTPI - PDISTPI) > PDISTPI$
$NearDist$	$VDISTGOAL$	ADD	$PDESDIST$	LST	$PBESTDIST$
					$(VDISTGOAL + PDESDIST) < PBESTDIST$
$FarDist$	$VDISTGOAL$	SUB	$PDISTPI$	GRT	$PDISTPI$
					$(VDISTGOAL - PDISTPI) < PDISTPI$

Tabela 7.7: Exemplo de operações e comparações realizadas para ativar as transições.

$Comp_{ID}$	Significado		
EQ	$Fator1$	$=$	$Fator2$
NEQ	$Fator1$	$\neq$	$Fator2$
GRT	$Fator1$	$>$	$Fator2$
LST	$Fator1$	$<$	$Fator2$
GEQ	$Fator1$	$\geq$	$Fator2$
LEQ	$Fator1$	$\leq$	$Fator2$
AEQ	$ABS(Fator1)$	$=$	$Fator2$
ANEQ	$ABS(Fator1)$	$\neq$	$Fator2$
AGRT	$ABS(Fator1)$	$>$	$Fator2$
ALST	$ABS(Fator1)$	$<$	$Fator2$
AGEQ	$ABS(Fator1)$	$\geq$	$Fator2$
ALEQ	$ABS(Fator1)$	$\leq$	$Fator2$
EQA	$Fator1$	$=$	$ABS(Fator2)$
NEQA	$Fator1$	$\neq$	$ABS(Fator2)$
GRTA	$Fator1$	$>$	$ABS(Fator2)$
LSTA	$Fator1$	$<$	$ABS(Fator2)$
GEQA	$Fator1$	$\geq$	$ABS(Fator2)$
LEQA	$Fator1$	$\leq$	$ABS(Fator2)$
AEQA	$ABS(Fator1)$	$=$	$ABS(Fator2)$
ANEQA	$ABS(Fator1)$	$\neq$	$ABS(Fator2)$
AGRTA	$ABS(Fator1)$	$>$	$ABS(Fator2)$
ALSTA	$ABS(Fator1)$	$<$	$ABS(Fator2)$
AGEQA	$ABS(Fator1)$	$\geq$	$ABS(Fator2)$
ALEQA	$ABS(Fator1)$	$\leq$	$ABS(Fator2)$

Tabela 7.8: Opções de comparações disponíveis nos testes.

$Oper_{ID}$	Significado		
SUB	$Fator1$	$-$	$Fator2$
ADD	$Fator1$	$+$	$Fator2$
MUL	$Fator1$	$*$	$Fator2$
DIV	$Fator1$	$/$	$Fator2$

Tabela 7.9: Opções de operadores disponíveis nos testes.

Fase de Origem	Fase de Destino	$Teste_{ID}$
init	AlignSline	TRUE
Goal	disconect	stopped
AlignSline	GoToGoal	aligned
AlignSline	Goal	sucess&!colision
AlignSline	stopping	colision
GoToGoal	AlignWall	obstacle
GoToGoal	Goal	sucess&!colision
GoToGoal	stopping	colision
AlignWall	GetAway	aligned
AlignWall	Goal	sucess&!colision
AlignWall	stopping	colision
GetAway	GetBack	limitIdist
GetAway	Goal	sucess&!colision
GetAway	stopping	colision

Tabela 7.10: Exemplo de transições de fase do modelo.

Os Testes de Transição são utilizados pelo projetista para definir a condição necessária para ativar a transição de fase. A lista com os teste utilizados no protótipo pode ser vista no Apêndice D.

Uma transição é composta essencialmente da fase de origem e a fase de destino, juntamente com condição, definida por uma equação lógica na qual os fatores são os identificadores dos Testes de Transição. O formato de descrição de uma transição é o seguinte:

Origem Fase de origem.

Destino Fase de destino.

Equação Equação booleana que unifica as condições definidas. As equações podem se utilizar os seguintes operadores:

& – And

+ – Or

! – Not

***** – Xor

() – Parênteses para controlar as prioridades.

O formato das transições e condições associadas foi criado de forma a ter um processamento simples, além de permitir um alto grau de expressividade para o projetista. Como os testes são de natureza simples, é possível interpretá-los, ou simplesmente codificar automaticamente uma função na linguagem *C* que implemente os testes de forma eficiente.

Da maneira que as transições são definidas, é possível que duas transições distintas sejam habilitadas simultaneamente. Assim sendo, é necessário criar critérios de desempate ou de prioridade. Como a prioridade deve ser definida pelo projetista, é fácil assumir seqüência de definição das transições como o critério de prioridade entre elas. Em outras palavras, quando duas ou mais transições ficam ativas simultaneamente, a primeira na ordem que o projetista definiu é efetivada.

7.1.3 Fases equivalentes

Um sistema tolerante a falhas deve cumprir seu objetivo utilizando os recursos disponíveis da melhor maneira possível. Quando um defeito é diagnosticado e identificado fica mais simples reconhecer as limitações do sistema e selecionar a ação corretiva mais adequada. Infelizmente, o problema de diagnóstico não é simples, principalmente se tratando de sistemas robóticos que interagem com o mundo real. Pode ser muito difícil identificar um pneu furado, ou uma roda deslizando na lama, ou até mesmo uma câmera de vídeo tentando criar imagens no meio de neblina.

Os robôs atuais e provavelmente os futuros possuem um espaço de observação proporcionado pelos sensores muito menor que o espaço de falhas, principalmente se este incluir a interação com o meio. A realização de um diagnóstico refinado pode ser muito difícil ou totalmente inviável em função das poucas informações disponíveis.

A dificuldade em realizar um diagnóstico preciso abre espaço para outras abordagens em relação à recuperação de falhas. Quando o sistema percebe que alguma tarefa não esta sendo realizada adequadamente, este pode optar por tentar realizá-la de forma diferente, mesmo sendo incapaz de identificar a falha que o está atrapalhando. A vantagem desta abordagem indireta é permitir a recuperação de falhas mesmo que o defeito não seja identificado. A principal desvantagem é o tempo que o sistema pode levar para se recuperar de uma falha: o sistema pode demorar a perceber que a tarefa ou função não esta sendo realizada apropriadamente, pois depende da avaliação do efeito de suas ações no ambiente; o sistema pode realizar várias tentativas infrutíferas até que selecione uma ação que funcione.

A abordagem indireta foi utilizada no robô submarino criado por Payton et al. [Payton et al., 1992], no qual comportamentos com funções redundantes eram associados a índices de sucesso. Foi considerado importante favorecer no modelo, este tipo de recuperação de falhas, principalmente por esta ser complementar e não exclusiva, em relação à detecção e recuperação de falhas direta. Para incluir no modelo a

possibilidade de recuperação de falhas de forma indireta é necessário tratar dois pontos distintos: disponibilizar no controle alternativas diferentes para realizar a mesma tarefa no caso de falhas; incluir um mecanismo para detectar que uma tarefa não esta sendo realizada adequadamente.

A capacidade de um robô possuir múltiplas estratégias para realização de uma mesma tarefa é um requisito simples de ser atendido, quando se utiliza uma arquitetura híbrida com uma máquina de estados no alto nível. Ações alternativas para uma tarefa podem ser facilmente incluídas no nível mais alto do controle através da inserção de novas seqüências de fases no autômato finito que define uma missão.

O segundo requisito para a recuperação indireta é a inserção no modelo de um mecanismo para detectar que uma tarefa não esta sendo realizada adequadamente. Para ser facilmente empregado este mecanismo deve ter uma forma padrão, a qual foi definida como sendo um índice de sucesso de execução para a fase ($I_{\text{Fase}}^{\text{Sucesso}}$). Para maior integração ao modelo, este índice deve ser normalizado possuindo o valor na faixa de [0 até 1], além disso, ser atribuído a um *ED* do sistema ($ED_{I^{\text{Sucesso}}}$) por um *BF* desenvolvido pelo projetista.

O índice de sucesso é definido especificamente para cada fase da missão, permitindo o desenvolvimento de funções apuradas para cada situação do sistema. A Tabela 7.11 mostra os atributos de fase associados com o processo de recuperação indireta. A cada fase de uma missão é associado um *BF* responsável pelo calculo do índice de sucesso específico, desta forma, o projetista mantém o controle completo da função utilizada para avaliar a eficácia das ações efetuadas pelo robô.

Dependendo da necessidade do cálculo do índice, o *BF* pode executar em uma freqüência igual ou inferior a execução do ciclo da fase. Esta informação é útil se a demanda de processamento da fase for alta em relação às restrições temporais existentes. Uma freqüência menor pode ser facilmente implementada se alternando fluxos distintos que incluam ou não o calculo do índice. Por exemplo, uma freqüência de $\frac{1}{3}$ é implementada se alternando uma execução de um fluxo que inclua o calculo do índice com dois que não incluam. Este mesmo mecanismo pode ser empregado no calculo dos outros índices utilizados no controle adaptativo e de falhas.

O índice de sucesso sendo atribuído a um *ED* pode ser utilizado no Teste de Transição normal de uma fase para outra, sem diferença em relação à de qualquer outro teste utilizado para transições. Este recurso já permite a implementação da recuperação de falhas indireta, entretanto requer que o projetista defina previamente todas as seqüências de fase com as tentativas de recuperação.

A recuperação indireta de falhas é realizada essencialmente através da tentativa de várias alternativas diferentes para realizar a mesma ação ou tarefa, até que uma delas alcance um índice de sucesso satisfatório. Para otimizar este processo reduzindo o tempo de recuperação são necessários alguns cuidados.

A realização de tentativas repetidas ou fechamento de ciclos neste processo em um

Atributo	Descrição
BF de Sucesso	Identificador do *BF* responsável pelo cálculo do Índice de Sucesso ($I^{Sucesso}$).
Freqüência	Freqüência de execução do *BF* de sucesso relativa a execução da fase ($0 < \text{freq} \leq 1$).
Sucesso Mínimo	Valor mínimo do Índice de Sucesso para a fase ser considerada viável ($0 <$ Sucesso Mínimo ≤ 1).
Sucesso Desejável	Valor mínimo do Índice de Sucesso para o sistema considerar uma fase alternativa. ($1 \geq$ Sucesso Desejável $\geq$ Sucesso Mínimo).
Tempo de avaliação	Tempo mínimo entre a ativação de uma fase e o cálculo do Índice de Sucesso ser considerado valido. Este tempo é necessário para avaliar a reação do ambiente as ações realizadas na fase. Pode ocorrer uma mudança de fase antes que o índice seja considerado válido.
Máximo de tentativas	Número máximo de tentativas que o controle tenta executar uma fase antes de desistir. Cada tentativa é contada após o Índice de Sucesso ser avaliado abaixo do Sucesso Mínimo.
Tempo entre tentativas	Tempo mínimo entre duas tentativas utilizando a mesma fase, considerando que a ultima execução teve um Índice de Sucesso menor que o Sucesso Mínimo.

Tabela 7.11: Atributos relacionados com a recuperação indireta de falhas de uma fase.

Fases equivalentes:		F1, F2, F3	
Fase origem	Fase equivalente	Fase de preparação	Prioridade
F1	F2	-	0
F1	F3	FP13	10
F2	F1	-	0
F2	F3	FP23	0
F3	F1	-	10
F3	F2	-	0

Tabela 7.12: Exemplo de um conjunto de fases equivalentes.

curto espaço de tempo pode ser prejudicial à recuperação ao sistema, pois o mesmo pode consumir recursos em tentativas infrutíferas. Se existem múltiplas alternativas de recuperação, deve-se começar primeiramente com as de maior probabilidade de sucesso. A ordenação das tentativas pode envolver tanto a analise da confiabilidade associada a cada uma, quanto o histórico recente de tentativas. A implementação de múltiplas ordenações de tentativas através de máquina de estados finitos é complexa, pois pode requerer a inclusão de todas as combinações de seqüências simultaneamente.

No modelo desenvolvido, a melhor solução encontrada foi embutir na *PCA* e em suas bibliotecas de funções o controle de recuperação indireto. O projetista define quais são as fases de uma missão equivalentes entre si. Duas fases equivalentes entre si realizam a mesma tarefa de formas diferentes, podendo inclusive utilizar recursos de hardware distintos. O projetista pode associar a cada fase um número máximo de tentativas e um índice de Sucesso Desejável, abaixo do qual é ativada uma transição para uma fase equivalente (Tabela 7.11).

A alteração entre duas fases equivalente pode requerer uma etapa extra de ajuste do sistema, a qual é representada por uma fase de preparação. A transição entre as fases de preparação e a fase equivalente deve ser especificada como uma transição normal de fase. Como mostrado na Tabela 7.12, o projetista também pode definir um critério de prioridade entre as transições.

A *PCA*, para implementar a recuperação indireta, mantém o histórico do último índice de sucesso associado a cada fase [1] juntamente com o número de tentativas infrutíferas já realizadas, e o momento em que foram realizadas. Quando uma fase esta apresentando um índice de sucesso menor que o mínimo estabelecido, a *PCA* seleciona uma alternativa que atenda os seguintes requisitos.

- Existe uma fase alternativa equivalente.

[1] O índice de sucesso de cada configuração distinta de uma fase (Seção 7.3.2) também é armazenado, permitindo que as transições de fase sejam realizadas entre as configurações que apresentam um melhor resultado.

- A confiabilidade da fase alternativa é maior que o mínimo definido (Seção 6.5.1).
- A fase alternativa não alcançou o número máximo de tentativas infrutíferas.
- Já decorreu o tempo mínimo entre as tentativas da fase alternativa.

Quando existir disponível mais de uma fase equivalente atendendo os requisitos, é necessário um conjunto de critérios para selecionar a tentativa de recuperação. Estes são enumerados a seguir:

1. A fase equivalente cuja última tentativa seja a mais antiga. Este requisito força com que todas as fases equivalentes disponíveis sejam tentadas.
2. A fase equivalente possui índice de sucesso desconhecido ou o maior índice de sucesso entre todas as alternativas disponíveis. O índice de sucesso é calculado durante a execução do fluxo e, portanto não pode ser previsto. Este critério favorece uma alternativa ainda não tentada ou se todas já foram tentadas, a que obteve maior sucesso.
3. O projetista pode definir uma ordem inicial ou uma ordenação parcial para as tentativas, de forma a direcionar o processo de recuperação indireta.
4. A fase equivalente possui índice de confiabilidade maior entre todas as alternativas disponíveis. Como a confiabilidade é calculada pelo sistema de diagnóstico associado, esta seleção evita o uso de configurações prejudicadas por defeitos ou falhas já identificadas.

Quando o projetista define para uma missão seu conjunto de fases, incluindo as equivalentes, define também as alternativas para alcançar o sucesso desta. Se for associado a cada uma das fases um índice de sucesso é possível identificar valores mínimos e máximos deste índice para cada possível caminho. Os valores obtidos podem ser significativos para representar o índice de sucesso esperado da missão completa. Um índice global de cada missão poderia ser utilizado na seleção de tarefas em times cooperativos, como visto no trabalho [Dias and Stentz, 2000].

O calculo dos índices de sucesso de qualquer ação realizada por um robô, associado no modelo as fases e a missão é um extremamente complexo, sendo o mesmo tema, de muitos trabalhos na literatura. Abordagens estáticas muitas vezes se mostram inadequadas ou ineficazes quando são consideradas missões de maior duração, nas quais os desgastes ou ajustes do sistema interferem na interação com o meio. Existem atualmente abordagens que ajustam os parâmetros de sucesso esperados dinamicamente, inclusive para o controle de times cooperativos como os trabalhos de Parker [Parker, 1999]. A forma do cálculo do índice, juntamente com as suas implicações aumentaria muito o escopo do trabalho e por isso foi deixado para trabalhos futuros (Seção 10.1).

7.1.4 Fases de Recuperação

Em sistemas complexos como os robôs, o diagnóstico de um defeito e a seleção de uma ação adequada para recuperar de uma falha pode ser um processo demorado em relação ao dinamismo existente. Em muitos casos, é necessário que o sistema adote medidas preventivas enquanto não identifica corretamente a fonte de um problema. Em outros, pode ser também necessário iniciar ações específicas para aumentar o espaço de observação do sistema e facilitar ou refinar o processo de diagnóstico. Em qualquer uma destas situações a tarefa corrente que o robô esta realizando é interrompida.

No modelo desenvolvido a interrupção da ação corrente, que corresponde a uma fase da missão corrente, é implementada através da inclusão de Fases de Recuperação. Estas podem ser associadas diretamente à missão como um *default* para todas as fases ou associada especificamente a cada fase normal.

Mission:	GOTOGOAL
Default_recovery_phase:	GRADUAL_STOP_PHASE
recovery_phase:	FAST_STOP_PHASE(PHASE1,PHASE2,...PHASEn)

Uma Fase de Recuperação têm o objetivo de manter ou levar o sistema a uma situação segura minimizando a chance de provocar danos a ele mesmo ou ao ambiente, enquanto garante ao sistema de diagnóstico tempo suficiente para procurar uma configuração confiável para o estado atual de falhas.

As Fases de Recuperação são implementadas como as outras fases normais do modelo podendo inclusive participar de transições de fase. Quando uma fase é definida como de recuperação, ela é obrigada a possuir um *BF* que produza um valor para um *ED* específico do sistema (ED^{Rec}). Este *ED* realiza a integração de uma Fase de Recuperação e a *PCA*, podendo assumir os seguintes valores mostrados na Tabela 7.13.

A integração das Fases de Recuperação no controle de alto nível é extremamente simples. Basicamente são inseridas transições especiais das fases normais para as Fases de Recuperação e transições das Fases de Recuperação para a fase de *Recuperação Global*. As transições de retorno para as fases normais não são necessárias, pois o contexto de uma missão pode ser mantido internamente a *PCA* quando é executado um processo de recuperação. Quando uma Fase de Recuperação é executada o *PC* de sistema PC^{Status}_{system} fica com o valor *RECOVERY*, permitindo ajustes internos aos *BFs* do fluxo se necessário. Um exemplo de transições pode ser visto na Figura 7.2.

A definição de uma Fase de Recuperação deve incluir o mínimo de processamento possível, ou seja, poucos e *BFs* simples. A troca de contexto para uma Fase de Recuperação deve representar um impacto mínimo no limite de tempo de cada ciclo. Caso

NULL	O controle híbrido mantém na Fase de Recuperação ou realiza uma transição de fase normal.
ABORT	O controle híbrido desiste da missão e realiza uma transição para uma fase de *Recuperação Global* (Seção 7.4).
PHASERECOVER	O controle híbrido tenta retornar para a fase que estava executando antes de ser desviado. O controle pode selecionar uma configuração diferente do fluxo de processamento que apresente uma confiabilidade maior para a mesma fase. Caso não seja encontrada uma configuração viável é realizada uma transição para a fase de *Recuperação Global*, interrompendo assim a missão.
MISSIONRECOVER	O controle híbrido tenta retornar para uma fase equivalente a fase que estava executando antes de ser desviado. Caso não seja encontrada uma configuração viável é realizada uma transição para a fase de *Recuperação Global*.
RECOVER	O controle híbrido tenta retornar para a fase que estava executando antes de ser desviado. Caso não encontre uma configuração viável para mesma fase, este tenta encontrar uma fase equivalente viável. Caso nenhuma das tentativas seja possível é realizada uma transição para a fase de *Recuperação Global*.

Tabela 7.13: Valores do ED^{Rec} utilizado para controle de uma fase de recuperação.

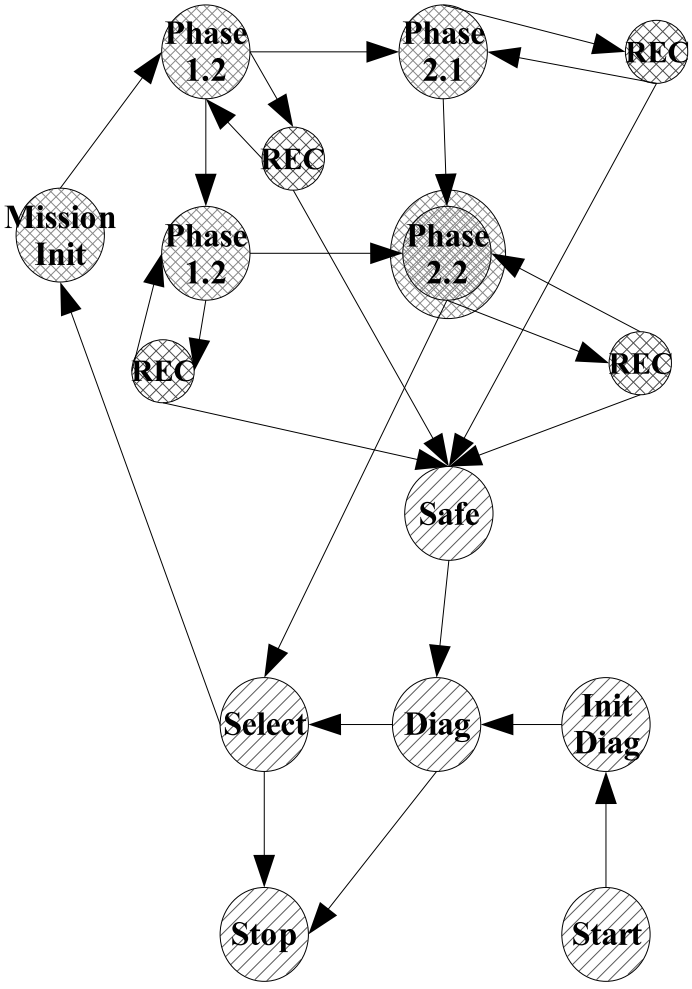


Figura 7.2: Exemplo de uma missão incluindo as Fases de Recuperação.

os limites de tempo sejam muito restritos, pode-se criar uma seqüência de fases normais iniciando em uma de recuperação para permitir ao sistema etapas sucessivas de adequação. Durante as fases de recuperação testes de detecção de falhas específicos, juntamente com as funções responsáveis pelo diagnóstico do sistema podem ser executadas, de forma a obter as informações necessárias para seleção da ação corretiva mais adequada.

A ativação de uma Fase de Recuperação é realizada quando é detectada uma nova falha, como é visto na Seção 8.2, e o sistema o sistema não é capaz de selecionar imediatamente uma fase que recupere do erro encontrado.

7.2 Configuração do controle adaptativo

O pessimista queixa-se do vento.
O otimista espera que ele mude.
O realista ajusta as velas.

Willian George Ward
(1812-1992)

Quando um sistema possui tolerância a falhas adaptativa se conclui que no mesmo, existem várias políticas de redundância ou configurações para cada tarefa específica, e estas políticas são selecionadas de acordo com o seu estado e objetivo atual. No modelo desenvolvido, cada política de redundância corresponde a um fluxo específico de processamento de baixo nível, com a configuração de *PCs* associada. Falta definir o processo de seleção da configuração mais adequada a um dado momento.

Uma configuração é adequada, se realizar a tarefa a que se propõe da melhor forma possível. Infelizmente, é necessário para o controle adaptativo selecionar uma configuração sem a executá-la, ou seja, baseando-se em previsões ou estimativas. Estas previsões devem levar em conta dois fatores: a probabilidade de executar a tarefa sem erros (I^C); e a eficiência na sua execução (I^D). Estes dois fatores devem ser combinados criando um único índice para se comparar configurações diferentes, sendo este um fator de ganho (I^G).

7.2.1 Índice de confiança

Uma tarefa pode não ser realizada por um robô por muitos motivos. Pode existir um ou mais defeitos no sistema, ou o ambiente apresenta propriedades não previstas que estejam interferindo com a tarefa, ou o controle não foi programado adequadamente.

Definir um índice com a probabilidade de sucesso ou falha na execução de uma

tarefa de um robô é um problema complexo, o qual só gera muitos trabalhos interessantes na literatura ([Parker, 2000b]). Neste trabalho, como foi visto na Seção 6.5, foi realizada uma abordagem simplista, na qual é calculada a probabilidade do fluxo gerar uma informação em um atuador ou em outro *ED*, com dados provenientes de uma falha.

Os objetivos de uma fase podem ser traduzidos para o índice de confiança de uma determinada configuração (I^C), através da associação de pesos a confiança das informações atribuídas a determinados *EDs*, como visto na Equação 6.5.6 na Página 109 da Seção 7.1.1.

O cálculo realizado levou em conta apenas a probabilidade de falhas nos dados coletados, não considerando a influencia direta da precisão destes na probabilidade de sucesso da fase. Este ponto é deixado como trabalhos futuros (Seção 10.1).

Uma abordagem interessante para obter este índice, considerando não apenas as falhas, mas também a precisão das informações é executar as múltiplas configurações do sistema e coletar os dados de sucesso. Podem-se inserir falhas facilmente nos *EDs* de forma a validar ou ajustar os cálculos de confiança realizados de forma estática. Infelizmente, esta abordagem pode ser bastante trabalhosa ou inviável, se o número de configurações e a quantidade de pontos de inserção de falhas for muito grande, o que normalmente acontece.

7.2.2 Índice de desempenho

A medida de desempenho na realização de uma tarefa pode corresponder ao tempo e aos recursos demandados. Intuitivamente, associamos que quanto mais rápido é executada uma ação, maior é o seu desempenho. Mas é importante destacar que o consumo de um recurso como energia pode ser tão importante quanto o tempo decorrido.

A inferência de um valor de desempenho, para a realização de uma tarefa, é tão complexo para um robô quando a inferência da sua confiabilidade ou do sucesso na realização da mesma.

Da mesma forma que a obtenção da confiança, uma abordagem mais simples, mas muito trabalhosa, seria executar cada possível configuração do sistema e coletar os dados relevantes referentes ao seu desempenho, obtendo assim índices médios e limites referentes a cada uma. Se o número de configurações for muito grande, pode-se tentar selecionar um subconjunto que caracterize as variações com impactos significativos no desempenho, obtendo neste caso, dados suficientes para permitir a inferência do desempenho das configurações não avaliadas. Um modelo de desempenho é desenvolvido baseado no conhecimento do funcionamento do sistema e em dados experimentais coletados.

Para facilitar a manipulação, o índice pode ser normalizado entre $0 < I^D \leq$

1. Essencialmente, a configuração com melhor desempenho, ou que consome menor tempo, recebe o índice com valor 1, e as demais são calculadas proporcionalmente, como é visto na Equação 7.2.1.

$$I_c^D = \frac{min(T_k) \forall k \in \text{Adaptações}}{T_c} \tag{7.2.1}$$

Fica claro que o ajuste deste índice pode ser um processo lento e muito interativo. No modelo desenvolvido o fator de custo mais claro é o processamento demandado pelo fluxo de baixo nível e a memória ocupada. Como foi visto em vários momentos no texto todo o trabalho foi desenvolvido buscando a concentração de processamento no fluxo. Uma simplificação possível no modelo para um índice de desempenho é associá-lo ao custo de processamento do fluxo. Para tanto é necessário o tempo de processamento de cada *BF*, que pode ser obtido em execuções reais ou simulações com entradas controladas. Além disso, é necessário o tempo consumido em trocas de contexto e configurações realizadas internamente a *PCA*. Estes valores podem ser obtidos com instrumentação do sistema em situações controladas como visto na Seção 6.8.

A configuração que consome menor processamento definido pela soma de tempo dos *BFs* recebe o índice com valor 1. As demais são calculadas proporcionalmente como é visto na Equação 7.2.2.

$$I_c^D = \frac{min(\sum_{i \in BFs^k} T_i) \forall k \in \text{Adaptações}}{\sum_{i \in BFs^c} T_i} \tag{7.2.2}$$

De forma a buscar uma maior aproximação com a realidade ou permitir um refinamento da estimativa de desempenho, é possível tentar combinar o tempo de processamento do fluxo com outras informações internas, por exemplo, valores armazenados em configurações de *PCs* associadas à fase. Se a tarefa da fase for percorrer um ambiente, a sua velocidade média pode ser um fator importante de desempenho independente do tempo total da missão ou do tamanho do ambiente. Se a velocidade média variar em função da fase ou do período do ciclo de processamento, como foi visto na Seção 6.4, o desempenho da configuração sofre uma influencia direta desta variação. Assim sendo, a estimativa de desempenho pode ser calculada considerando este tipo influencia, como visto na Equação 7.2.3, na qual a velocidade foi incluída na formula de calculo.

$$I_c^D = \frac{min((\sum_{i \in BFs^k} T_i) \times PC_{Vel}^k) \forall k \in \text{Adaptações}}{(\sum_{i \in BFs^c} T_i) \times PC_{Vel}^c} \tag{7.2.3}$$

Entre cada fase do controle a ação realizada pode variar significativamente, assim como os fatores que influenciam o desempenho final observável. Devido a estes motivos, uma estimativa do índice de desempenho calculado em função de parâmetros

internos ao controle também deve se adequar a cada fase. O projetista deve poder associar a cada fase de uma missão, a equação apropriada para o cálculo da estimativa de desempenho I^D.

PHASE:	GOTOGOAL
IPERF(I^D):	CYCLE_PCVEL
MIN_IPERF:	0.50

Este cálculo do índice pode ser implementado em uma forma equivalente a linguagem de definição de testes definida na Seção 7.1.2, ou através de uma função na linguagem *C*, que receba como parâmetro o identificador da formula de cálculo. Independente da forma de implementação, o cálculo deve ter acesso a dados internos da *PCA* e informações armazenadas na estrutura do Grafo de Controle Adaptativo (*GCA*) para poder criar uma estimativa adequada.

No protótipo desenvolvido, com será visto na Seção 9.2 o número de configurações era pequeno sendo possível executar cada uma individualmente. O tempo médio de execução foi calculado e o índice de desempenho de cada configuração foi estabelecido manualmente.

7.2.3 Índice de ganho

Em determinada missão de um sistema pode ser mais importante o desempenho, e em outra a confiabilidade, refletida no modelo pela confiança. Normalmente estes dois fatores são antagônicos, pois a confiabilidade depende do uso de redundância a qual provavelmente vai consumir uma quantidade maior de recursos.

Se a confiabilidade de determinada ação é muito baixa, pode não fazer sentido tentar realizá-la, porque provavelmente o sistema irá falhar. Da mesma forma pode ser inviável, se tentar realizar uma ação que demore muito tempo ou consuma toda a bateria ou outro recurso disponível.

O controle adaptativo visa balancear a confiabilidade com o desempenho, sendo que, a importância de cada fator a um dado momento deve ser estabelecida de forma externa a este controle. Seja definida por um operador humano ou por um sistema de planejamento capacitado para tal. Em qualquer caso, uma função correspondente ao ganho do sistema I^G deve ser definida, e o objetivo do controle adaptativo passa a ser maximizar o seu valor. A função de ganho deve combinar a importância de dois ou mais indicadores do sistema, visando criar um índice único utilizável em um processo de seleção da configuração mais apropriada a um dado momento. Como exemplificado na Equação 7.2.4, o índice de ganho pode ser definido por uma função entre I^D, I^D e $F_{\text{Importância}}^{\text{Desempenho}\times\text{Confiança}}$ que é um fator de balanceamento de importância entre os dois anteriores.

$$I_c^G = f_G(I_c^D, I_c^C, F_{\text{Importância}}^{\text{Desempenho}\times\text{Confiança}}) \tag{7.2.4}$$

O processo adaptativo implementado consiste basicamente na seleção de um nodo vizinho no Grafo de Controle Adaptativo, que será visto na Seção 8.1, que possua um I^G maior que o nodo corrente. Neste caso, uma adaptação é realizada. A função f_G pode ser composta de várias formas diferentes, por exemplo, através de uma função linear como mostrado na Equação 7.2.5, que está sendo utilizada no protótipo.

$$I_c^G = (I_c^D \times F) + (I_c^C \times (1 - F)) \tag{7.2.5}$$

A função linear é um exemplo que utiliza um único fator para definição da importância $F^{Desempenho\times\text{Confiança}}$ e fornece um resultado normalizado com foi proposto também pelos índices I^C e I^D. A função de ganho pode ser tão complexa quanto o projetista queira, mas a essência, sempre será balancear a importância de fatores antagônicos. Se os fatores a serem balanceados não forem antagônicos, no sentido que um melhora, e outro piora, não a necessidade de uma função de ganho deste tipo. No controle adaptativo podem ser incluídos outros fatores para otimização explicita, como consumo de bateria ou combustível, bastando para isso, ampliar a formula de ganho.

MISSION:	M_GOTOGOAL
MIN_IPERF:	0.30
MIN_IREL:	0.70
MIN_IPROFIT:	0.90
PHASE:	F_GOTOGOAL
IPERF(I^D):	CYCLE_PCVEL
MIN_IPERF:	0.50
IREL(I^C):	STDCALC
MIN_IREL:	0.80
IPROFIT(I^G):	STDLINEAR
MIN_IPROFIT:	1.00

Como já citado anteriormente, no modelo criado, é importante a possibilidade de definir os índices do controle adaptativo de forma global a missão, ou associados diretamente a cada fase específica. Desta forma, é possível respeitar as características

próprias de cada missão, assim como de cada fase. O refinamento ao nível de fase, permite que o projetista ajuste individualmente a importância dos fatores adaptativos, de forma a garantir critérios coerentes com o objetivo global da missão. Sendo o fator $F^{\text{Desempenho}\times\text{Confiança}}$ definido globalmente para a missão, pode ser necessário em determinadas fases valorizar a confiança mais que o desempenho e vice versa. Além disso, deve ser possível especificar os níveis mínimos de desempenho ou confiança aceitáveis, abaixo dos quais a fase ou missão é considerada inviável.

O ideal para o uso do modelo, é que as formulas básicas sejam identificadas e inseridas automaticamente através de um ambiente específico de desenvolvimento (*framework*), e que os índices de controle utilizados sejam aprendidos dinamicamente a partir do funcionamento do robô. Esta nova etapa do trabalho, que facilitaria em muito o trabalho do projetista, fica também para o futuro (Seção 10.1).

7.2.4 Custo de uma adaptação

Uma questão importante que não foi tratada neste trabalho é quando a ativação de uma configuração realizada no processo de adaptação está associada a um custo, seja este de tempo, energia ou outra natureza. A ativação de uma nova configuração de software no modelo implementado [2] possui uma troca de contexto mínima, a qual está associada somente, à mudança de índices internos aos *PCs* automáticos.

Quando uma adaptação de um sistema está associada à ativação ou desativação de elementos de hardware, existe um custo no processo de reconfiguração. Este custo pode ser em energia, tempo, recursos de processamento ou outros. Assim sendo, o custo das adaptações também deve ser inserido no cálculo de ganho. Essencialmente, a função de ganho passa incluir um fator temporal, que é o tempo esperado de funcionamento na nova configuração. A inclusão do tempo esperado na nova configuração e o custo de transição insere basicamente uma inércia no processo adaptativo.

Este problema é equivalente a trabalhos de outras áreas, por exemplo, a otimização do consumo de energia em sites de provedores web ([NETO et al., 2003]), no qual servidores podem ser ativados ou desativados em função da demanda. Esta questão, assim como outros pontos, foram deixados para trabalhos futuros (Seção 10.1).

7.3 Controle do processamento no fluxo

A cada fase do controle de alto nível deve ser definida a funcionalidade específica do controle de baixo nível. Para tanto, o projetista define um conjunto de *BFs* que realizam o fluxo de processamento desejado para cada fase, implementando o

[2] Esta observação é válida em sistemas monoprocessados e multiprocessados no qual não existe *overhead* na comunicação entre processos de processadores distintos.

processamento da percepção e seleção das ações do robô. Além disso, o projetista deve selecionar uma ou mais configurações de *PCs* adequadas ao controle da fase, como visto na Seção 7.1.1.

Por analogia a outras arquiteturas híbridas, podemos dizer que cada tarefa ativa um conjunto de comportamentos, ou um conjunto de esquemas motores e perceptivos, ou um conjunto de equações dinâmicas de controle. No modelo não importa muito qual é a abordagem de baixo nível escolhida, e sim a forma de implementação. Qualquer uma delas é factível com nenhuma ou poucas alterações na forma de comunicação dos *BFs*. Por exemplo, se na implementação dos *EDs* fossem criadas mensagens de inibição e supressão seria muito simples implementar o modelo de comportamentos em baixo nível do Brooks [Brooks, 1989b].

7.3.1 Configurações dos Parâmetros de Controle

Um objetivo constante na área de computação é reaproveitamento ou compartilhamento de código já escrito. Isto pode ser feito através da programação estruturada e orientada a objetos ou com o uso de camadas genéricas (*APIs*). Um dos fatores que facilitam a reutilização de código no modelo é o uso dos Parâmetros de Controle.

Como foi visto na Seção 6.4, os Parâmetros de Controle multivalorados foram inseridos no modelo com intuito de facilitar a configuração do nível mais baixo do controle, permitindo o ajuste interno aos *BFs* de forma externa a eles. A função principal dos *PCs* automáticos é oferecer ao sistema um grau a mais de liberdade ou flexibilidade para as adaptações. Os valores dos *PCs* devem refletir as adaptações do controle e do fluxo internamente nos *BFs*.

Os Identificadores de Configuração associam e indexam valores aos *PCs*, como pode ser visto na Tabela 6.4 da Página 102 e na Tabela 6.6 da Página 103. Para cada fase de uma missão deve-se associar um Identificador de Configuração, instruindo o controle sobre quais os valores devem ser lidos dos *PCs* em qualquer configuração da fase, como pode ser visto no Tabela 7.14.

7.3.2 Definição dos *BFs* associados a fase

Um robô deve sempre saber o que fazer com seus atuadores, mesmo que seja ficar parado ou continuar a ação corrente. No modelo desenvolvido, este requisito transposto para o fluxo de baixo nível significa que em cada ciclo de processamento valores devem ser atribuídos aos *EDs* associados aos atuadores do robô.

O projetista deve associar para cada fase um conjunto de *BFs* responsável pelo processamento de baixo nível. Como é visto na Seção 6.2.2, a informação topológica existente nas interconexões dos *BFs* é suficiente para se gerar um escalonamento válido. Se um atuador não será utilizado em determinada fase da missão, o projetista

Atributo da fase	Valor
MISSION:	M_GOTOGOAL
DEFAULT_PC_IDCONF:	**CONF01**
PHASE:	F_GOTOGOAL
PC_IDCONF:	**CONF02**
.. PHASE:	F_GOTOGOAL
PC_IDCONF:	**CONF02**
PHASE:	F_GETAWAY
PC_IDCONF:	**CONF03**
...	

Tabela 7.14: Exemplo de associação de Identificadores de Configuração com fases da missão.

pode opcionalmente atribuir um valor fixo ao *ED* associado ao atuador, de forma a estabelecer um valor *default* para este. Também opcionalmente, o projetista pode identificar *EDs* atuadores ou não que não influenciam na fase corrente, ou seja, o sucesso da execução da fase corrente não depende destes elementos.

O projetista pode definir todos os *BFs* necessários para o processamento da percepção até a ação da fase. Mas é possível que ele defina apenas os *BFs* responsáveis pela atuação da fase, deixando o processamento da percepção parcialmente ou completamente indefinido. A Figura 7.3 exemplifica este conceito. O projetista associa a fase da missão um conjunto de *BFs* essenciais e as transições. As transições estão associadas a Testes, os quais por sua vez estão associados a *PCs* e *EDs*. Os *BFs* definidos podem possuir *EDs* de entradas que não correspondam a sensores. Assim sendo, a definição parcial dos *BFs* de uma fase juntamente com as transições de fase determinam um conjunto de *EDs* os quais devem ter o seu valor produzido pelo fluxo de dados, para que a fase seja executável.

Agregando sucessivamente ao conjunto, novos *BFs* que produzem os valores dos *EDs* necessários, é possível definir um caminho de processamento que conecte os *EDs* perceptivos aos *EDs* de atuadores, criando assim um fluxo completo de processamento para a fase. A Figura 7.4 mostra uma visão geral do fluxo neste processo. Caso exista redundância, podem ser gerados múltiplos conjuntos distintos de *BFs*. Cada um possui características diferentes de desempenho ou confiabilidade, correspondendo a possíveis configurações ou políticas de redundância do controle adaptativo.

A composição da estrutura oferece liberdade para utilizar múltiplas combinações ou alternativas de *EDs* e *BFs* para fornecer os valores dos *EDs* necessários pelos *BFs* associados às fases. Esta liberdade é utilizada para criar as reconfigurações necessárias

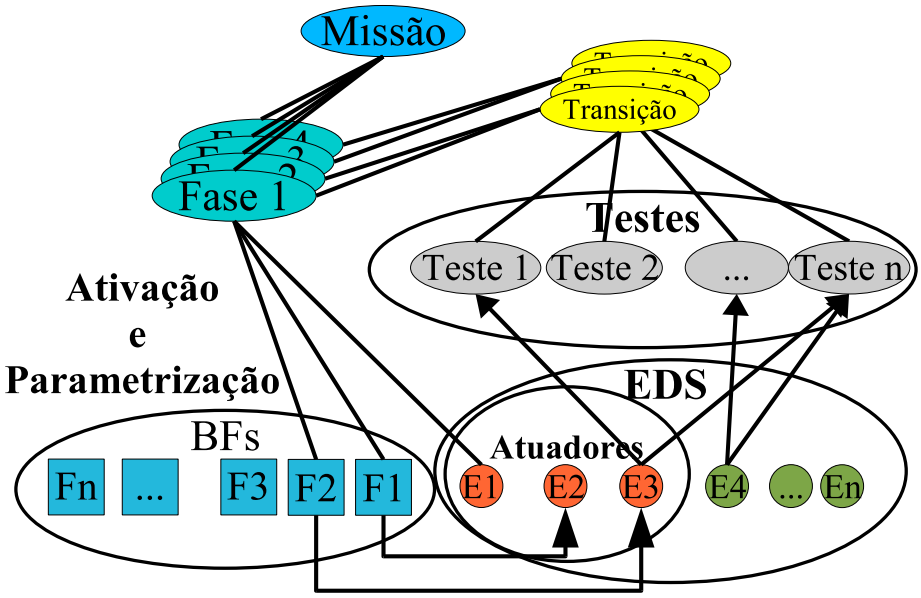


Figura 7.3: Estrutura de ativação de baixo nível por uma fase.

para implementação da tolerância a falhas nos sensores ou no processamento das informações.

Este processo de composição do fluxo é simples de ser implementado, pois a estrutura regular de definição do fluxo garante isto. O objetivo é formar todas as combinações possíveis de *BFs* que contenham os *BFs* já definidos pelo projetista conectados aos *EDs* de sensores e atuadores. Este processo de formação do fluxo pode ser facilmente automatizado, mas deve respeitar a algumas regras:

- Executar sempre os *BFs* definidos pelo projetista para a fase. Neste caso, gerar os valores dos *EDs* necessários para execução destes. Estes *BFs* são essenciais para uma fase.
- Gerar os valores de *EDs* utilizados nos testes de transição da fase.
- Respeitar as incompatibilidades de *BFs* definidas.
- Respeitar a unicidade de atribuição ao um *ED* associado a um atuador.

  Não é necessário gerar um valor para um atuador ao qual esteja associado um valor *default* para a fase.

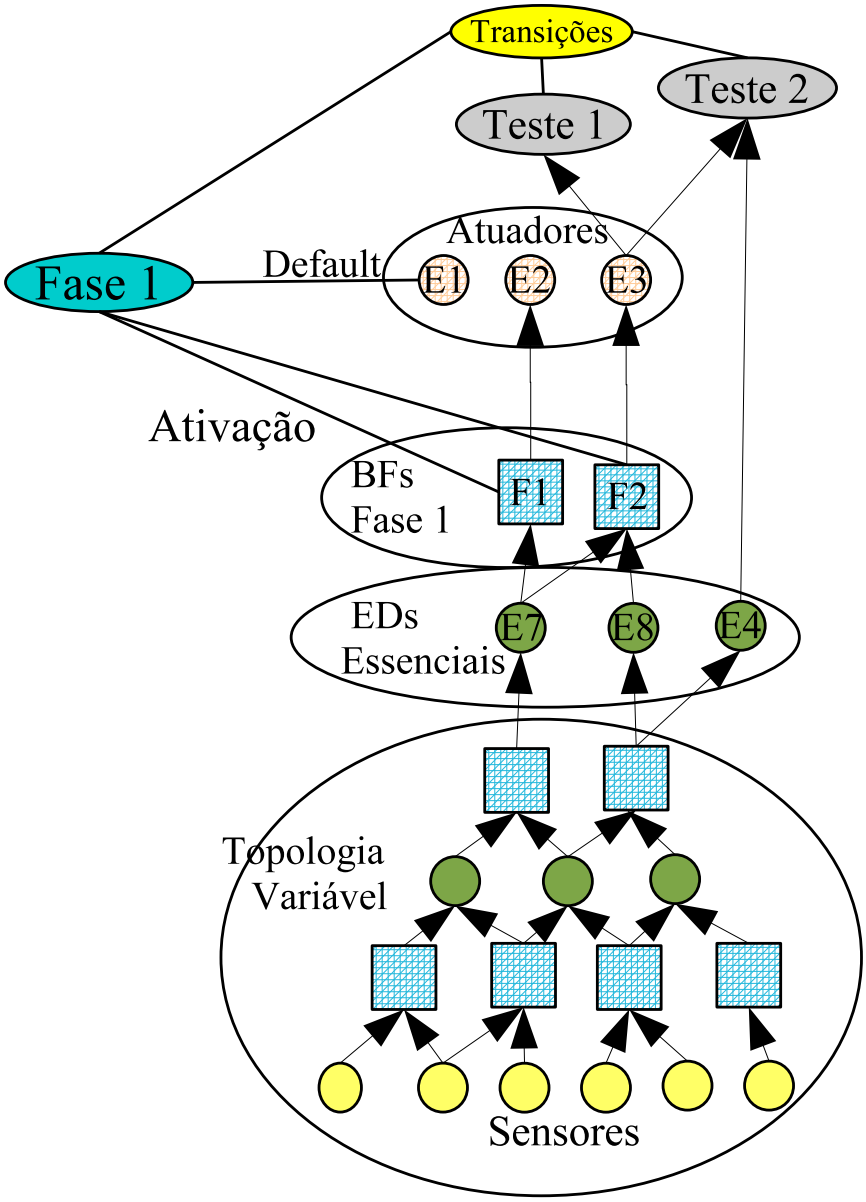


Figura 7.4: Estrutura de síntese do fluxo de processamento para uma fase.

Topologia	Descrição
IR	Utiliza apenas os sensores de proximidade por infravermelho.
SN	Utiliza apenas os sensores de proximidade por ultra-som (sonares).
IRSN	Utiliza simultaneamente os sonares e os sensores infravermelhos.
IRSNM	Utiliza simultaneamente os sonares, os sensores infravermelhos, e informações de distância provenientes de um mapa de obstáculos do ambiente construído dinamicamente.

Tabela 7.15: Configurações de subfluxos equivalentes existentes na topologia do protótipo implementado.

- Iniciar o fluxo nos *EDs* associados a sensores, indicadores internos do sistema, ou *EDs* de memória. O fluxo sempre deve acessar todos os dados necessários a sua execução completa.

- Gerar os valores dos *EDs* associados a *EDs* de memória utilizados como entradas no fluxo. Em outras palavras, se um *ED* de memória for utilizado no fluxo, o seu valor também tem que ser obrigatoriamente produzido.

Um ponto muito importante a se perceber, é que todos os possíveis caminhos existentes, foram definidos explicitamente na topologia de conexões dos *BFs* pelo projetista. Embora possam existir vários caminhos redundantes o número de possibilidades normalmente será pequeno para um sistema automatizado. Portanto, é possível utilizar inclusive algoritmos exaustivos. Além disso, os subfluxos gerados serão utilizados provavelmente por várias fases distintas, o que permite combiná-los facilmente no processamento de sistemas mais complexos. No desenvolvimento do protótipo esta etapa foi realizada manualmente, não sendo necessário o uso de algoritmos próprios.

No exemplo da Figura 7.5 é mostrada a topologia completa do protótipo desenvolvido. A redundância existente neste protótipo é apenas nas distâncias obtidas dos sensores de Infravermelho e Sonares. Existe ainda a possibilidade do uso de um mapa de obstáculos do ambiente, que oferece neste caso, uma redundância de informação de origem histórica. O mapa foi implementado utilizando simultaneamente as informações dos sonares e dos sensores infravermelhos, e por isso só faz sentido utilizá-lo com todos os sensores ativos. A Figura 7.6 exemplifica a redundância existente na informação de distância de obstáculos no protótipo. Neste caso, existem quatro topologias diferentes exemplificadas na Tabela 7.15 e na Figura 7.7.

O processo de construção do fluxo a partir dos *BFs* essenciais é mostrado utilizando o Exemplo 7.16, no qual o *BF"FollowWall"* é essencial. A Figura 7.8 apresenta a topologia inicial a fase F_GETAWAY da missão, na qual foram incluídos também os *EDs* necessários para os testes de transição. As figuras seguintes (Figura 7.9 e

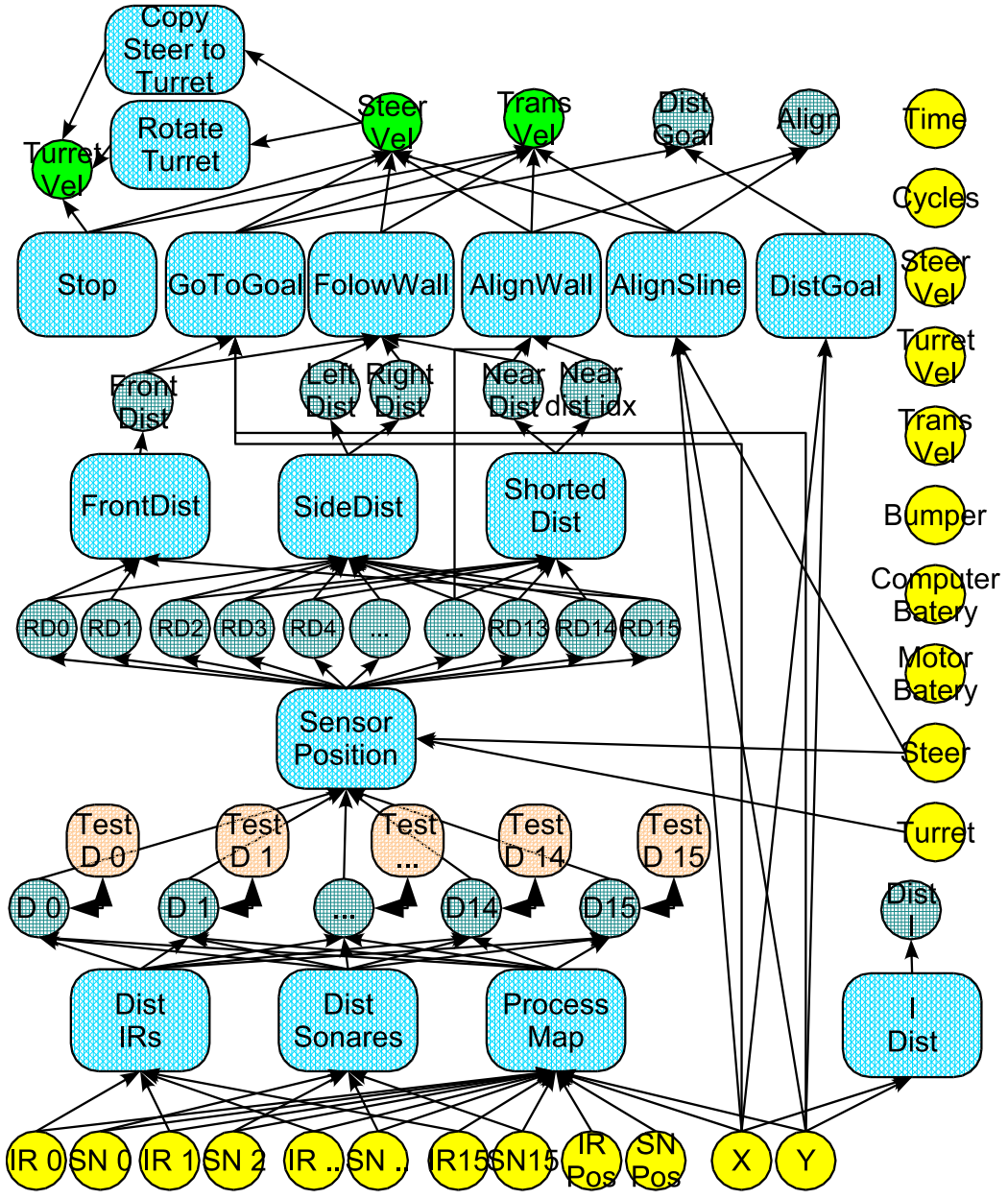


Figura 7.5: Topologia completa de conexão de *BFs* e *EDs* do protótipo.

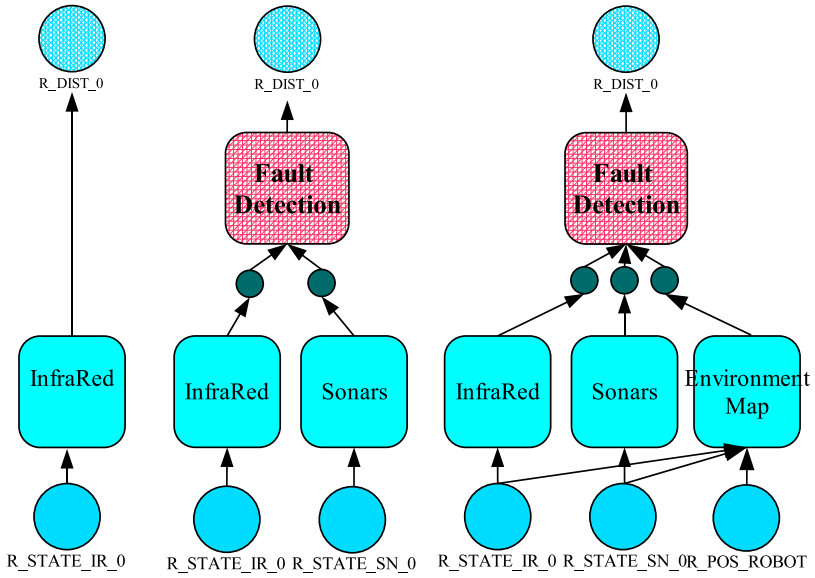


Figura 7.6: Exemplo de redundância existente no protótipo.

PHASE:	F_GETAWAY
ESSENTIAL_BFS:	FollowWall

Tabela 7.16: *BFs* essenciais de uma fase do protótipo.

Figura 7.10) apresentam etapas intermediárias da agregação sucessiva de novos *BFs*. A partir da topologia completa (Figura 7.5) e da definição inicial de *BFs* essenciais (Figura 7.8) para a fase é possível se obter quatro configurações distintas mostradas nas Figuras: Figura 7.11; Figura 7.12; Figura 7.13; e Figura 7.14.

Uma questão muito importante a ser notada é a ativação dos *BFs* de Teste associados aos *EDs* de distância que possuem informações redundantes. O teste implementado só faz sentido com dois ou mais valores redundantes, ou seja, associados às configurações IRSN e IRSNM. Quando os teste são ativados nestas duas configurações de fluxo são geradas mais duas possibilidades em um total de seis, como visto na Tabela 7.17.

No modelo existe a possibilidade de ativar individualmente cada um dos testes associados a cada *ED* de distância, o que proporcionaria um total de $2 + 2 * (2^{16})$ combinações diferentes de fluxo. Entretanto, ativar parcialmente os testes, só faz

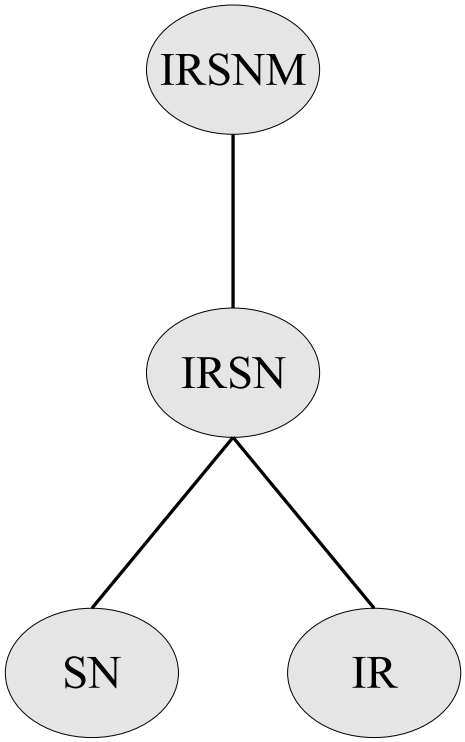


Figura 7.7: Conjunto de adaptações implementadas no protótipo.

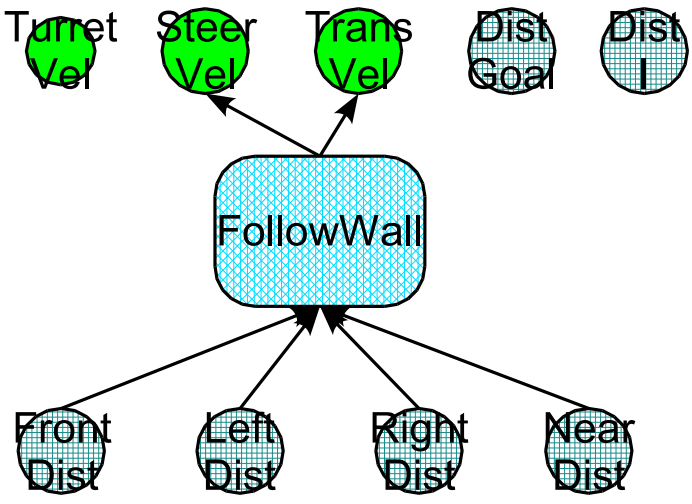


Figura 7.8: *BFs* essenciais e *EDs* para testes de uma fase.

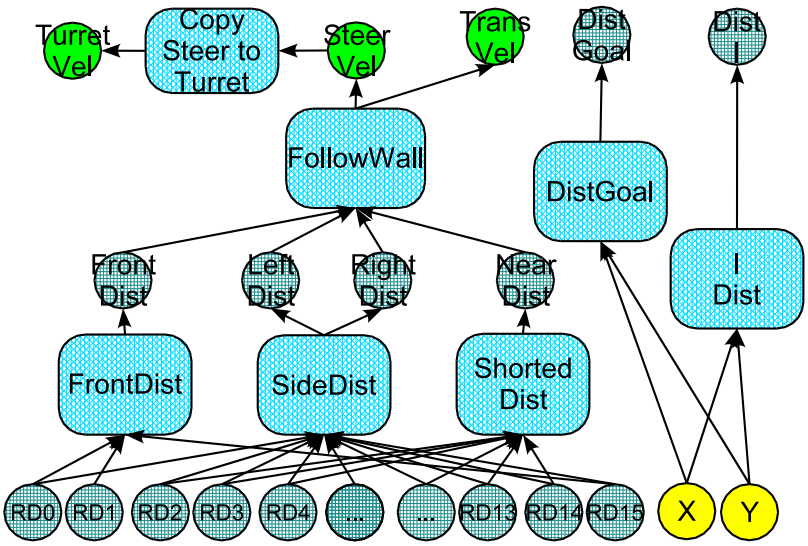


Figura 7.9: Primeira agregação de novos *BFs*.

Topologia	Descrição
IR	Utiliza apenas os sensores de proximidade por infravermelho.
SN	Utiliza apenas os sensores de proximidade por ultra-som (sonares).
IRSN	Utiliza simultaneamente os sonares e os sensores infravermelhos.
IRSNM	Utiliza simultaneamente os sonares e os sensores infravermelhos e informações de distância, provenientes de um mapa de obstáculos do ambiente construído dinamicamente.
IRSNT	Utiliza simultaneamente os sonares e os sensores infravermelhos. Os teste de detecção de falhas nos *EDs* associados às distâncias de obstáculos estão ativos.
IRSNMT	Utiliza simultaneamente os sonares e os sensores infravermelhos e informações de distância, provenientes de um mapa de obstáculos do ambiente construído dinamicamente. Os teste de detecção de falhas nos *EDs* associados às distâncias de obstáculos estão ativos. Esta configuração é mostrada na Figura 7.15.

Tabela 7.17: Configurações de equivalentes incluindo os testes de detecção de falhas.

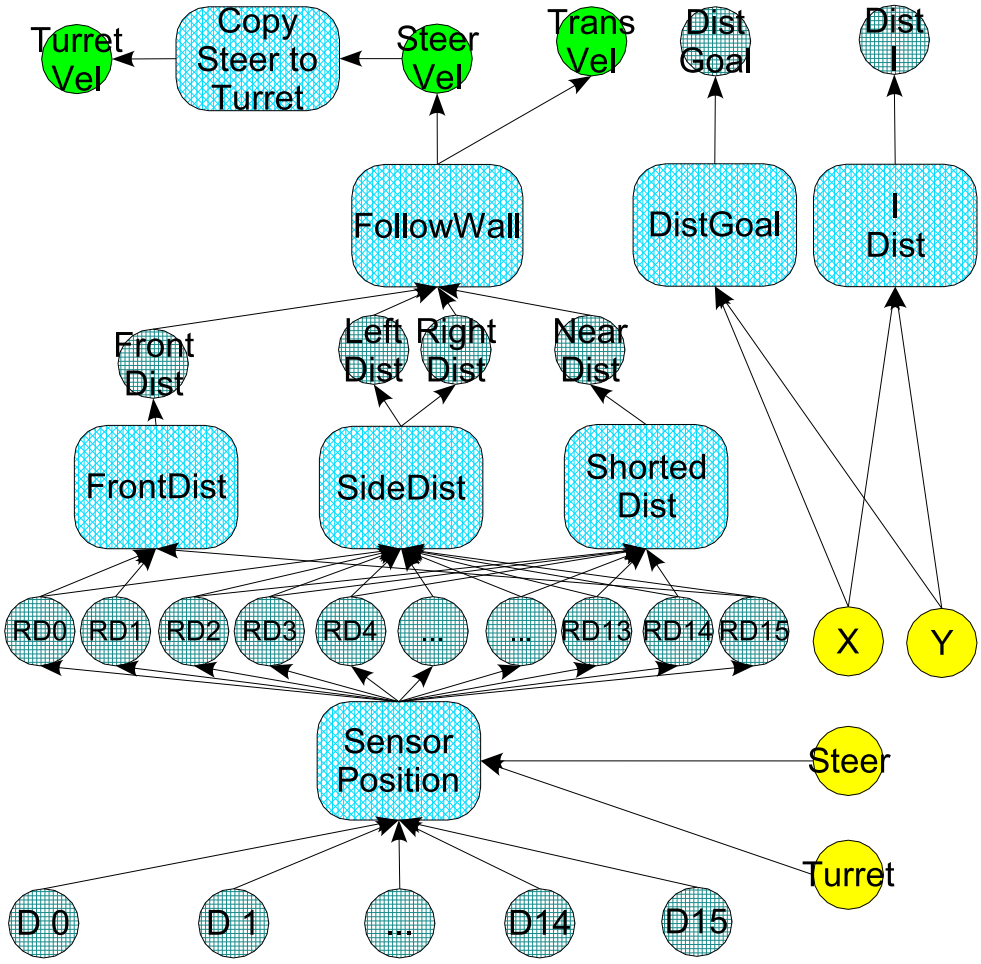


Figura 7.10: Segunda agregação de novos *BFs*.

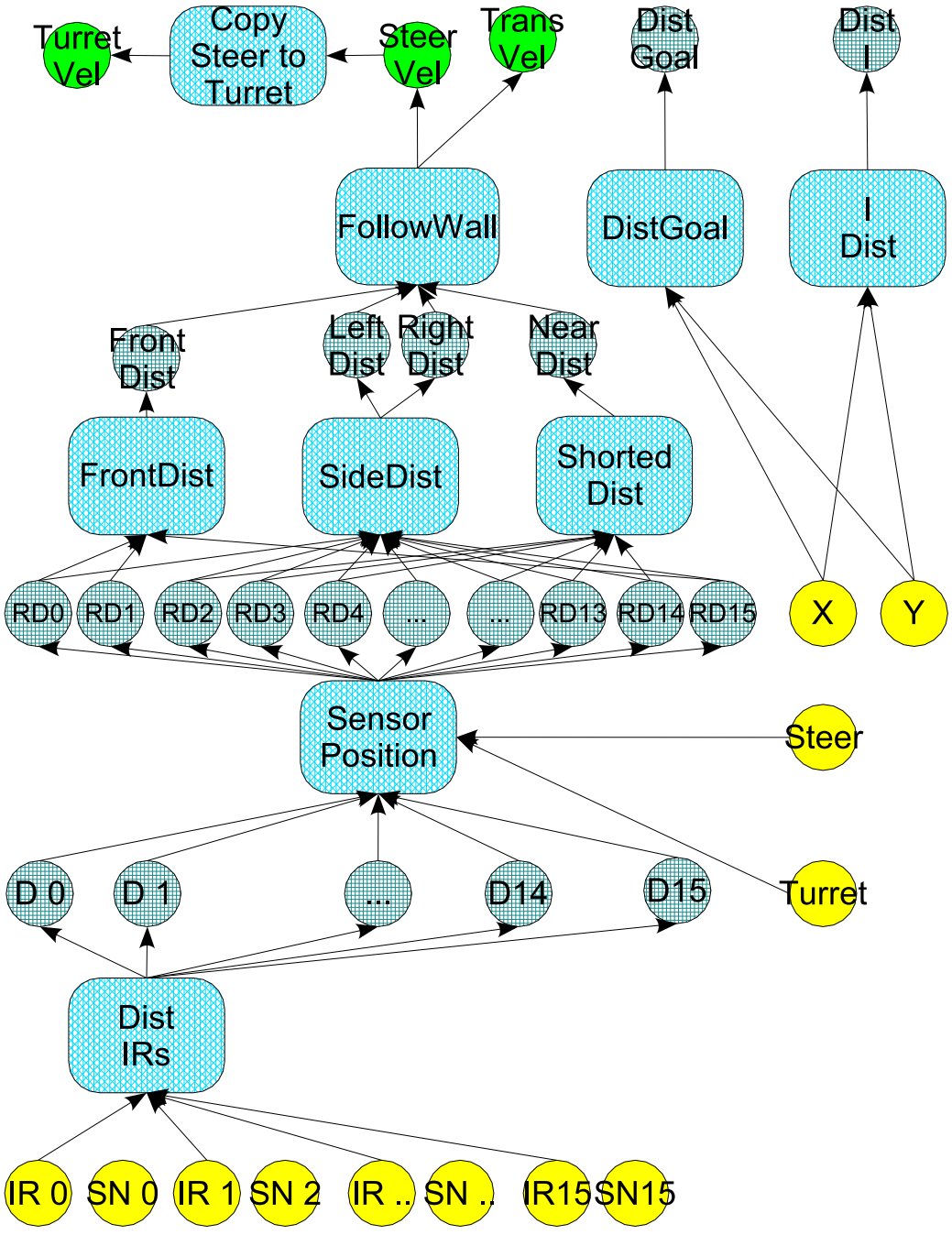


Figura 7.11: Fluxo completo utilizando sensores IR.

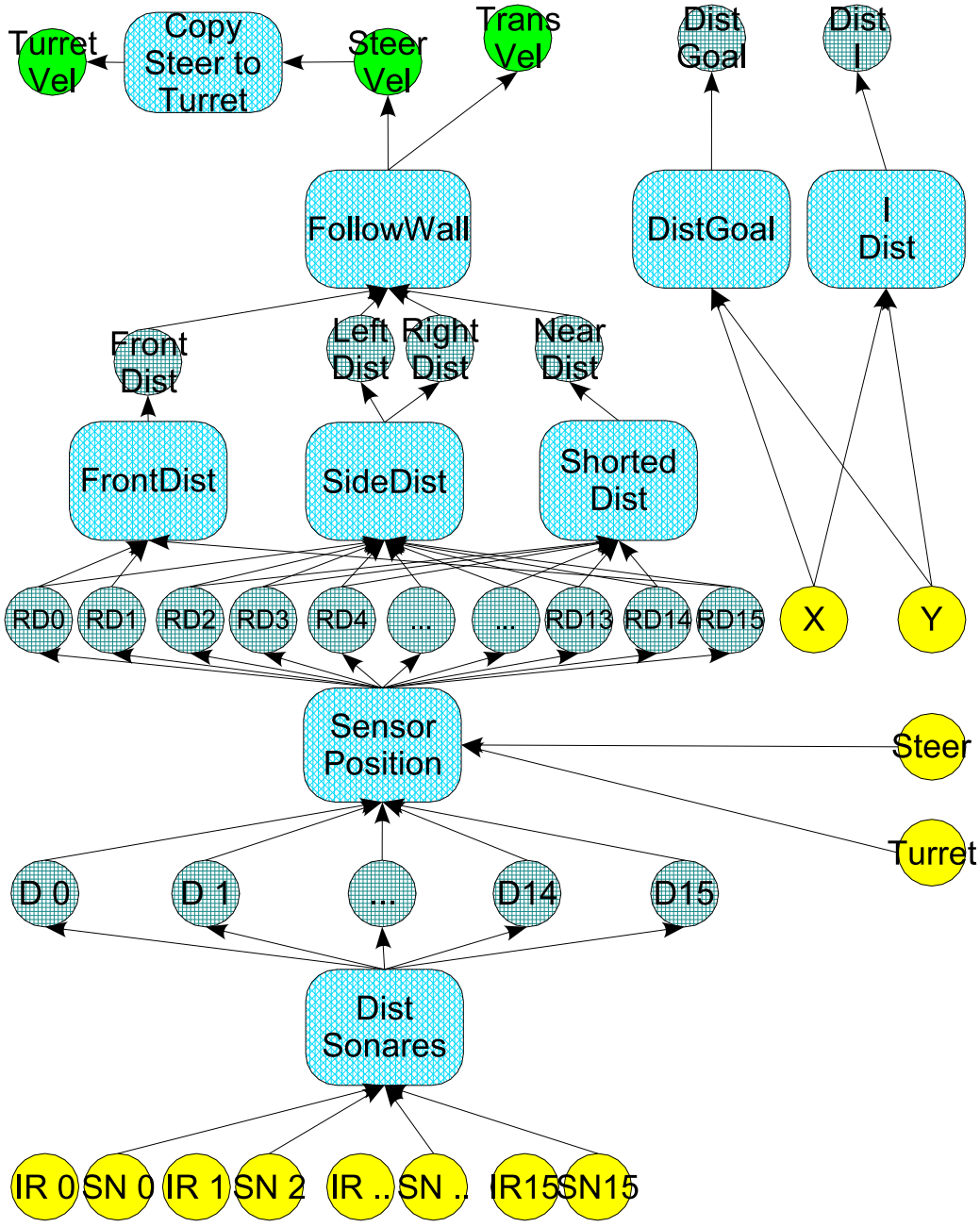


Figura 7.12: Fluxo completo utilizando sonares.

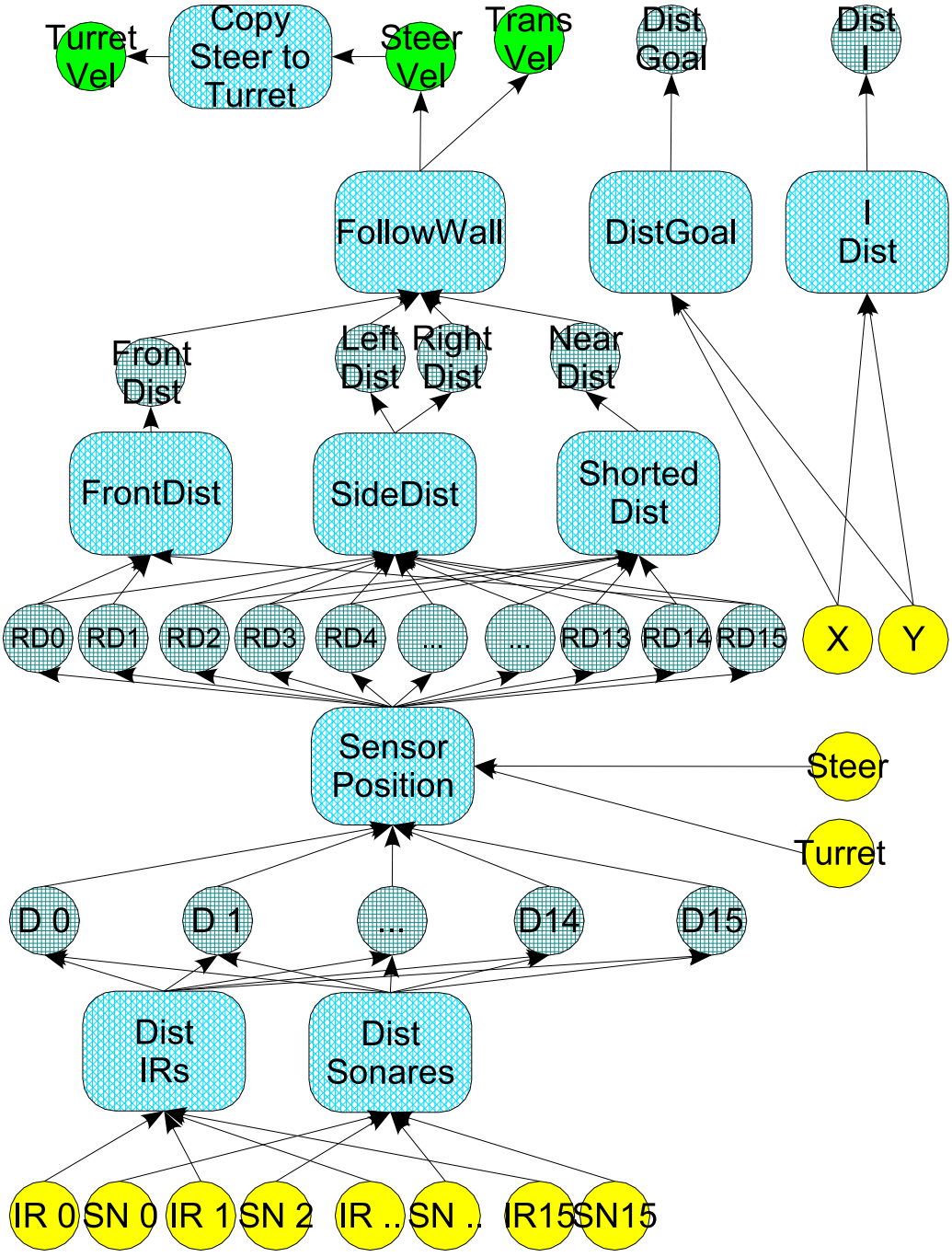


Figura 7.13: Fluxo completo utilizando sensores IR e sonares.

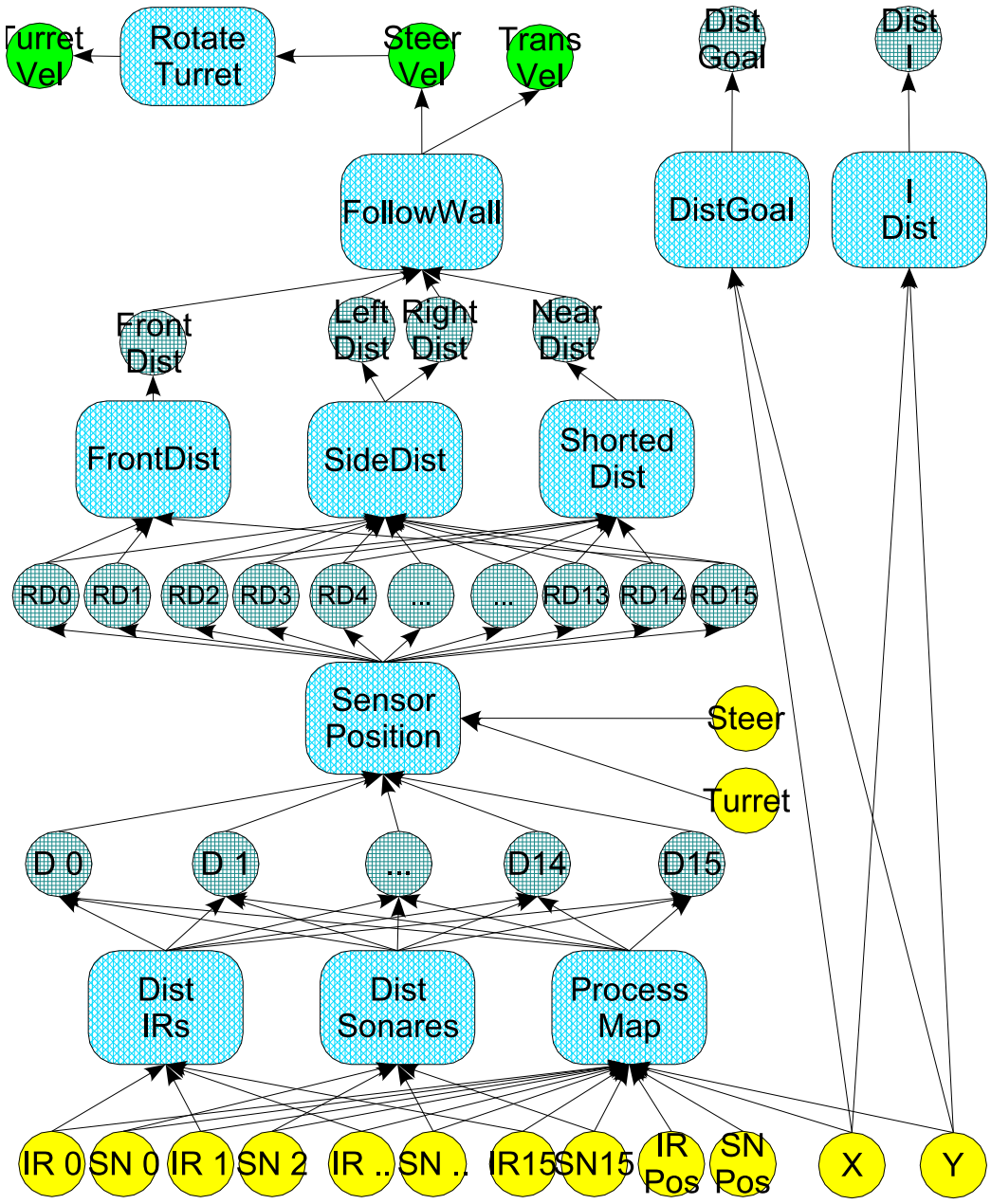


Figura 7.14: Fluxo completo utilizando sensores IR, sonares e um mapa do ambiente.

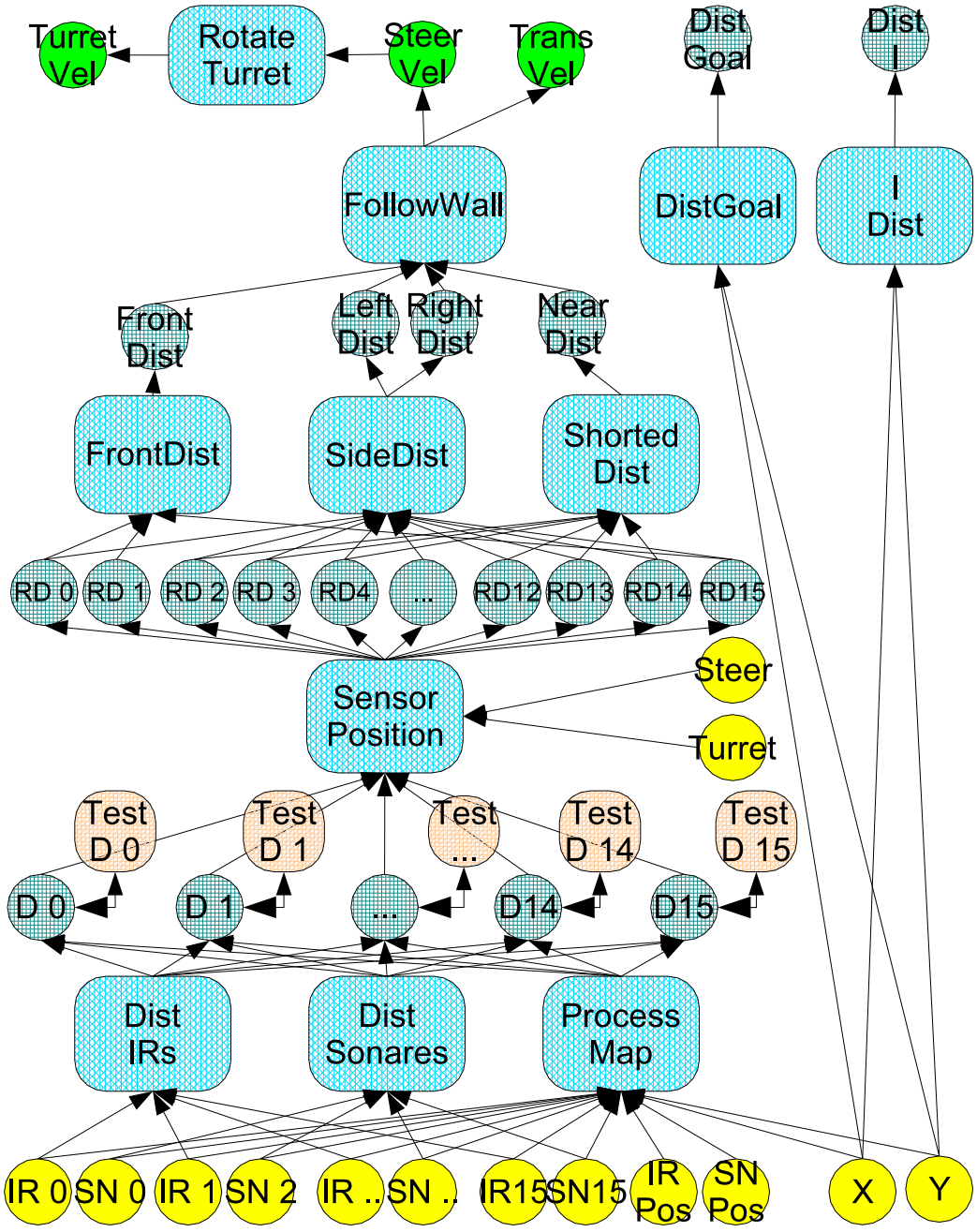


Figura 7.15: Fluxo utilizando sensores IR, sonares, mapa e testes de detecção de falhas.

sentido se a redução do custo de processamento em relação ao ganho de confiabilidade for significativa. Em topologias com vários pontos de redundância distintos e de natureza diferentes pode ser interessante esta liberdade. Por exemplo, quando um robô esta movimentando, os testes ativos podem ser referentes a esta ação, e quando ele estiver manipulando um objeto com a garra os testes interessantes podem ser outros. Vale lembrar que o cálculo da confiança associado à fase (Seção 6.5.1) pode levar estas variações da ação em consideração. Além disso, como foi dito nas Seções 6.3 e 6.1, podem ser incluídas nas descrições de *BFs* e *EDs*, restrições para controlar a ativação parcial dos testes, por um ambiente de desenvolvimento que crie as diferentes configurações automaticamente.

Cada configuração de fluxo de processamento que possui algum caminho diferente possui também características próprias de desempenho e confiabilidade. À medida que a redundância aumenta o custo de processamento também aumenta, pois a quantidade de informação processada é maior. No exemplo anterior do protótipo, fica claro que uma configuração de fluxo, representada pelo escalonamento [3] de *BFs*, pode ter muitos caminhos iguais se comparado a outros fluxos, podendo inclusive existir subconjuntos.

O modelo define também uma missão de sistema (Seção 7.4) que será integrada ao demais missões definidas pelo projetista. As fases desta missão especial também são implementadas por escalonamentos de *BFs*, que devem ser totalmente definidos. Neste caso, não ha possibilidade de se criar adaptações para estas fases específicas.

7.3.3 Definição das transições entre adaptações

Cada configuração sintetizada para uma mesma fase corresponde a uma possível adaptação diferente do sistema. Para que o mesmo se adeqüe de forma gradual a cada alteração do estado global é necessário criar uma estrutura de navegação entre as configurações. A forma mais simples e clássica de navegação é criar um grafo no qual os nodos representam as configurações e as arestas representam as possíveis trocas de configuração ou adaptações.

A definição dos nodos é bem clara, mas qual o melhor critério a ser utilizado para conectar as arestas. A opção mais simples é criar um grafo totalmente conectado, como mostrado na Figura 7.16. Como o controle adaptativo vai estar sempre avaliando outras configurações conectadas pelas arestas, um grafo completamente conectado pode oferecer um conjunto muito grande de opções. Este número pode impactar negativamente no desempenho do processo de seleção da adaptação, além de não garantir a realização de reconfigurações de forma gradual.

O processo adaptativo detalhado na Seção 7.2, é conduzido essencialmente pelos índices I^C e I^D. Assim sendo, uma segunda opção para criar as arestas é conectar os nodos próximos em função de ganho de desempenho ou ganho de confiança. Neste

[3] A representação de um fluxo na forma de um escalonamento de *BFs*, como visto na Seção 6.2

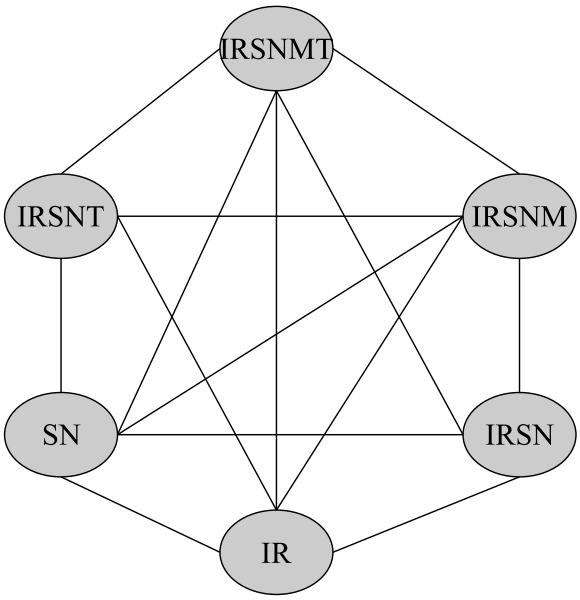


Figura 7.16: Grafo de configurações de uma fase do protótipo totalmente conectado.

caso, a primeira questão que surge é a definição de nodos próximos. A semelhança entre as configurações criadas através do agrupamento sucessivo de *BFs* baseado na topologia completa de interconexão é bastante evidente, como pode ser visto nas figuras: Figura 7.11, Figura 7.12, Figura 7.13 e Figura 7.14.

Os *BFs* normais de um fluxo possuem um identificador único e o mesmo pode ser considerado para um par constituído de um *BF* de teste e do *ED* associado. Neste caso, um fluxo pode ser definido de forma única por um conjunto C destes identificadores. Os nodos são representados pelo conjunto de todas as configurações $N_f = (C_1^f, ..., C_n^f)$ de fluxos geradas.

Com base no conteúdo de N_f, o processo de definição das arestas para atender o critério de proximidade desejado é bem simples, podendo ser feito em três etapas:

1. Um conjunto auxiliar de arestas é gerado conectando cada configuração a outras que sejam seu subconjunto. Um conjunto Φ de arestas é criado conectando o nodo i com o nodo j tal que:

   $$a_{i,j} \in \Phi \Longleftrightarrow (C_i^f - C_j^f) \neq \varnothing \wedge (C_j^f - C_i^f) = \varnothing \forall i, j \in N_f$$

2. Em um segundo passo, são removidas todas as adaptações que podem ser totalmente substituídas pela seqüência de duas adaptações distintas. Um conjunto ϵ de aresta é criado a partir de Φ tal que:

$a_{i,j} \in \epsilon \iff a_{i,j} \in \Phi \wedge \neg(a_{i,k} \in \Phi \wedge a_{k,j} \in \Phi \wedge (C_i^f - C_j^f) = ((C_i^f - C_k^f) \cup (C_k^f - C_j^f)))$

3. Se existirem *BFs* incompatíveis nas configurações, existe uma possibilidade do grafo gerado para as configurações da fase $G_f^A = (N_f, \epsilon_f)$ ficar desconexo. Neste caso, devem-se inserir arestas extras para conectá-lo. As arestas devem ser inseridas ligando nodos de subgrafos desconexos, entre as configurações com maior interseção de *BFs*. Seja Ψ e Ω dois conjuntos de nodos desconexos, as arestas extras devem ser inseridas em G_f^A tal que:

$$a_{i,j} \in \epsilon \iff i \in \Psi \wedge j \in \Omega \wedge (C_i^f \cap C_j^f) \subsetneq (C_k^f \cap C_l^f) \forall k \in \Psi \wedge \forall l \in \Omega$$

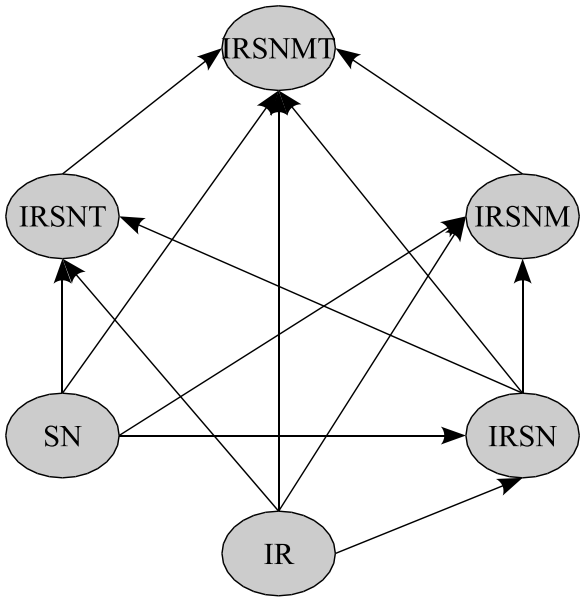


Figura 7.17: Grafo criado com as arestas em Φ.

O resultado da primeira etapa do processo de criação das arestas de adaptação, correspondendo ao conjunto Φ é mostrado na Figura 7.17. Após a remoção das arestas redundantes, se obtém o grafo apresentado na Figura 7.18 que corresponde ao conjunto ϵ.

Através da seqüência de passos descrita é gerado um grafo G_f^A conectando todas as configurações disponíveis para uma fase específica de uma missão. A forma de composição do grafo garante que as adaptações sejam feitas de mais gradual possível, além de garantir também a conectividade deste. Cada nodo do grafo fica conectado

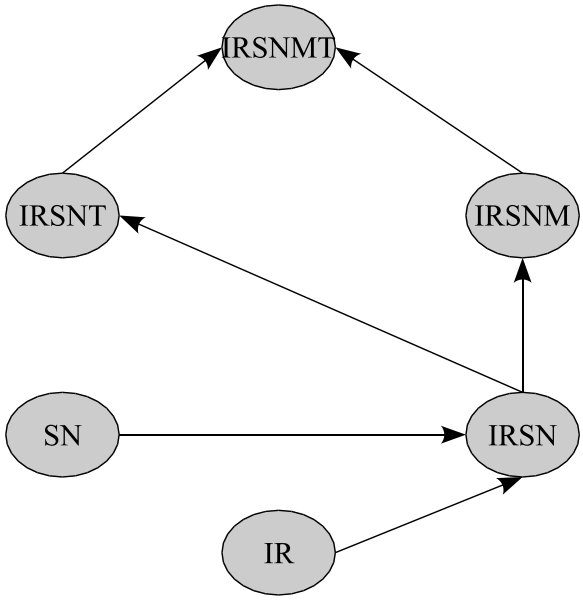


Figura 7.18: Grafo de adaptações final com as arestas em ϵ.

a configurações que contenham todos os seu *BFs* oferecendo um nível maior de confiança e fica conectado a configurações que correspondem a subconjuntos dos seus *BFs* podendo oferecer um nível melhor de desempenho.

A construção do grafo mantém implicitamente uma hierárquica de uso da redundância, e desta forma pode-se afirmar que a distância máxima entre dois nodos é $2 * (logn - 1)$, sendo n o número total de nodos. Comparando com uma árvore, o maior caminho existente seria de uma folha (redundância mínima) até uma outra, passando pela raiz (redundância máxima).

O controle adaptativo pode facilmente percorrer este grafo, pois as arestas sempre vão representar alterações de redundância. Além disso, são facilmente implementáveis buscas no grafo com a profundidade limitada, aumentando assim o escopo de seleção das adaptações. O processo adaptativo embora influencie na detecção e recuperação de falhas tem a finalidade principal de otimizar o uso dos recursos do sistema, o que torna plenamente aceitável, que seja feito progressivamente ao longo de vários ciclos de processamento.

7.4 Arestas de recuperação de falhas

As configurações geradas para cada fase, quando possuem caminhos redundantes no fluxo para a mesma informação, podem estar utilizando recursos distintos de hardware. Neste caso, estas configurações podem ser utilizadas para implementar a recuperação de falhas. A recuperação de uma falha é realizada quando uma nova falha é detectada e o sistema é reconfigurado para um fluxo que tolere a mesma.

O problema é que a recuperação de falhas deve ser realizada rapidamente, muitas vezes interrompendo o fluxo normal de processamento, e iniciando um novo conjunto de *BFs*. Para otimizar este processo a seleção de uma nova configuração deve ser previamente calculada. Da mesma forma que o controle adaptativo, pode ser construído um grafo contendo como nodos as configurações e como arestas as possíveis transições para recuperação de falhas.

As falhas que necessitam de uma recuperação específica são determinadas pelos *BFs* normais e *BFs* de teste que possuem a "*Ação de Recuperação de Falha*" definida na Seção 6.1 com ação "*Reconfigurar*". Estes *BFs* especificam que uma reconfiguração do sistema deve ser implementada se possível. Um teste associado a um *ED* detecta falhas em caminhos bem específicos. Assim sendo, se for inserida uma falha em um *ED* associado a um sensor, é possível identificar a sua propagação pelos *BFs* e pelos *EDs* de um fluxo específico até alcançar cada teste.

Um ponto claro é que a recuperação direta de uma falha só é possível após a detecção da mesma, ou seja, as arestas de recuperação de falhas vão estar associadas a testes específicos. Com estas premissas, o grafo de recuperação de falhas $G_f^R = (N_f, \phi_f)$ de uma fase é gerado da seguinte forma.

1. As arestas só existem a partir de configurações que possuam testes associados a reconfiguração. Assim sendo é definido um subconjunto Ψ com as configurações que atendam este requisito.

   $$c \in \Psi \iff \exists BF_j \in C_c^f | \text{Ação de Recuperação de Falha}_{BF_j} = \text{ Reconfigurar}$$

2. Para cada teste $BF_j \in C_c^f$ da configuração $c \in \Psi$ cuja "*Ação de Recuperação de Falha*"="*Reconfigurar*".

3. Para cada ED_i associado ao hardware, que iniciando a propagação de uma falha, pode ser detectada por BF_j. Ser detectada significa, ter a probabilidade de detecção maior que um mínimo. A confiabilidade dos *EDs* para propagação da falha é a mesma considerada na ausência de falhas, ou seja, como se o diagnóstico ainda não tivesse detectado a presença de um defeito no hardware.

   $$\text{Prob}^{BF_j}_{\text{Detect Fault}}(ED_i) > \text{Min}_{\text{Detect}}$$

4. Para o sistema se recuperar da falha em ED_i detectada por BF_j uma configuração que tolere a falha é procurada ($\forall k \in N_f$), que seja, se possível um superconjunto dos *BFs* da configuração atual e que possa ser executada em um tempo aceitável.

   $(I^C_{kf} \geq \min^{I^C_{\text{Fase}f}}) \wedge (C^f_c \subset C^f_k) \wedge (\text{Time}_{\text{exec}}(C^f_c \cup C^f_k) < \text{Time}^f_{\text{Limit}}) | \forall k \in N_f$

   A configuração k tolerar a falha no ED_i significa a mesma que tem um I^C (Seção 6.5) maior que o mínimo estabelecido para fase, mesmo que a confiabilidade do *ED* esteja abaixo de um mínimo para ser considerado defeituoso.

   $(r^{ED_i} < \min_{\text{Defeito}}) \wedge (I^C k^f \geq \min^{I^C_{\text{Fase}f}})$

   A configuração k ser um superconjunto dos *BFs* da configuração atual c significa que toda a redundância existente na atual, esta também contida em k.

   $C^f_c \subset C^f_k$

   A configuração k poder ser executada em um tempo aceitável, significa que o tempo de execução dos *BFs* de c e dos *BFs* de k, ainda é inferior ao limite máximo estabelecido para a fase f.

   $\text{Time}_{\text{exec}}(C^f_c \cup C^f_k) < \text{Time}^f_{\text{Limit}}$

5. Caso não seja encontrada nenhuma configuração que atenda o requisito de ser um superconjunto de c, este requisito é relaxado. Neste caso, a configuração com menor número de arestas separando c e k no grafo G^A_f é selecionada. Se mesmo assim, existirem várias configurações com o mesmo número de aresta, a configuração que apresentar um tempo de recuperação é selecionada.

   $(\text{Time}_{\text{exec}}(C^f_c \cup C^f_k) < \text{Time}^f_{\text{Limit}})$.

6. Quando uma configuração k atende os requisitos, uma aresta é inserida em ϕ_f contendo os atributos necessários.

   $\exists a_{c,k}(BF_j, ED_i) \in \phi_f$

Uma configuração que recupere de uma falha, deve apresentar um ou mais caminhos redundantes diferentes, da qual detectou a falha. Neste caso, obviamente se as duas configurações fossem unificadas, o número de *BFs* a ser executado seria maior que do fluxo atual, o que poderia violar restrições no limite de tempo para produzir os valores para os atuadores. Esta consideração deixa claro que a forma de recuperação de falhas vária em função da necessidade de adaptação do sistema.

Um teste (*BF* TA) associado a um *BF* pode detectar falhas de processamento ou de programação internas a ele. Neste caso, as soluções empregadas podem ser: uma nova execução da mesma função de processamento no mesmo processador ou em um processador diferente; ou a execução de uma função de processamento diferente com a

mesma funcionalidade, codificada de forma distinta. Se as restrições de tempo forem relaxadas, mecanismos internos a *PCA* podem resolver este problema de maneira simples, controlando o processo de execução. Se as restrições de tempo para o fluxo forem rígidas, será necessário criar configurações específicas para a recuperação da falhas de software. Para tratamento de falhas de processamento, a abordagem do modelo de usar um grafo para recuperação de falhas se mantém inalterada, mas novas funcionalidades nas estruturas de dados e na *PCA* devem ser incorporadas. Este ponto é deixado para os trabalhos futuros (Seção 10.1).

Como visto na Seção 7.2, cada configuração de uma fase gerada pode possuir valores diferentes para os índices adaptativos (I^C, I^D e I^G). Apenas para reforçar, pode-se realizar o cálculo de confiança de um fluxo utilizando as probabilidades de falhas conhecidas ou inferidas previamente dos elementos de hardware associados. Se uma configuração apresentar confiança, desempenho ou ganho inferior aos níveis aceitáveis, esta não é utilizada para recuperação de falhas podendo ser desconsiderada e removida do conjunto. Da mesma forma, se uma configuração for um subconjunto de outra, e seu desempenho for levemente superior, e a mesma não é utilizada para recuperação de falhas, esta configuração pode ser removida. O processo de remoção de configurações pode ser realizado até que, o projetista ou um sistema automatizado, obtenha um conjunto com diferenças de ganho significativas em relação aos possíveis estados de falhas do sistema.

Unificação dos autômatos de missões

Para simplificar o controle, é mais interessante existir apenas uma única máquina de estados ($M^{\text{robô}}$) para o robô, englobando todas as suas missões. Um autômato básico do sistema (M^{system}) é definido no modelo contendo fases de inicialização gerais e auxiliares. As fases principais de M^{system} são as detalhada na Tabela 7.18 e exemplificadas na Figura 7.19.

Os passos para se compor um autômato unificando as diversas missões definidas pelo projetista com a missão básica do sistema são as seguintes.

- Unificar todos as fases das missões com as fases M^{system}.

  $$Q^{\text{robô}} = Q^{\text{system}} \cup Q^1 \cup ... \cup Q^{m-1} \cup Q^m$$

- O conjunto de condições que ativam as transições compõe um conjunto único.

  $$\Sigma^{\text{robô}} = \Sigma^{\text{system}} \cup \Sigma^1 \cup ... \cup \Sigma^{m-1} \cup \Sigma^m$$

- A fase inicial de $M^{\text{robô}}$ corresponde à fase inicial de M^{system}.

  $$q_0^{\text{robô}} = q_0^{\text{system}}$$

Fase	Descrição
q_{Start}	Inicialização do sistema inclusive de aspectos internos da *PCA*.
q_{Diag}^{Init}	Inicialização do sistema de diagnóstico podendo inclusive iniciar uma seqüência de fases específicas para uma auto-analise do estado corrente do sistema.
q_{Select}	Realiza a seleção da missão sendo ativada por planejadores ou informações externas. Esta fase embute testes de viabilidade na execução de uma missão em função dos recursos disponíveis e dos defeitos existentes. Além disso, pode embutir o processo de seleção da adaptação com melhor ganho da nova missão selecionada.
q_{Safe}	Fase intermediária no qual o controle deve levar o sistema a uma situação na qual não provoque danos internos ou externos. Isto é realizado quando é detectada uma falha imprevista ou não recuperável. A próxima fase pode ser a parada completa ou uma nova etapa de diagnóstico. Esta fase corresponde a *Recuperação Global* definida na Seção 7.1.4.
q_{Diag}	Fase intermediária no qual o sistema fica parado em uma situação segura e pode iniciar uma nova missão de diagnóstico e reparos (M^{Diag}). Esta missão é importante para o projetista prover maneiras do sistema voltar à operação de forma supervisionada ou não. A realização de processos e ou ações específicas para aumentar as informações sobre o funcionamento do sistema pode permitir o refinamento do diagnóstico e a identificação mais precisa do defeito. Neste caso, o robô pode voltar à operação conhecendo as suas limitações correntes.
q_{Stop}	Estado final do controle sem atividades de atuação. Este estado deve ser ativado caso o robô receba um comando para tal, ou não possua mais a capacidade de realizar nenhuma de suas missões devido a restrições internas ou externas.

Tabela 7.18: Conjunto de fases auxiliares que conectam as diversas missões.

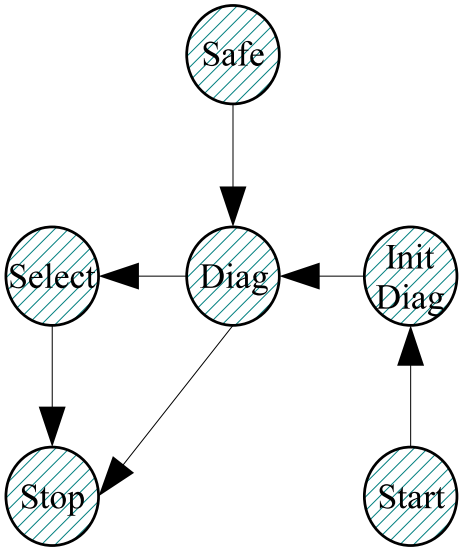


Figura 7.19: Conjunto de etapas auxiliares para unificação das missões.

- O conjunto de fases terminais de $M^{\text{robô}}$ corresponde ao conjunto F de M^{system}.

  $F^{\text{robô}} = F^{\text{system}}$

- O conjunto de transições de $M^{\text{robô}}$ correspondem a: união do conjunto de transições de cada missão (δ^i); transições da fase de seleção ($q^{\text{system}}_{\text{Select}}$) para fases iniciais de cada uma missão (q^i_0); transições dos estados finais de cada missão (F^i) retornando para a fase de seleção.

  $\delta^{\text{robô}} = \delta^{\text{system}} \cup \delta^1 \cup ... \cup \delta^{m-1} \cup \delta^m$

  $\cup(\forall i \in m(q^{\text{system}}_{\text{Select}} X \text{Select}^i) \rightarrow q^i_0)$

  $\cup(\forall i \in m(\forall q^j \in F^i(q^j X true) \rightarrow q^{\text{system}}_{\text{Select}}))$

Após a unificação das missões é obtido um autômato como todas as missões definidas pelo projetista $M^{\text{robô}} = (Q, \Sigma, \delta, q_0, F)$. Todas as fases e transições foram definidas pelo projetista ou foram previamente definidas para a missão M^{system}. É importante ressaltar que as fases com atributos especiais como recuperação ou equivalentes também estão presentes em Q.

Capítulo 8

Implementação

> Há dois tipos de pessoas: As que fazem as coisas, e as que dizem que fizeram as coisas. Tente ficar no primeiro tipo. Há menos competição.
>
> Indira Gandhi (1917-1984)

A implementação de todo o modelo definido no capítulo anterior é uma tarefa bem complexa. Tanto no aspecto de desenvolvimento da Plataforma de Controle Adaptativo quanto no de desenvolvimento de um ambiente que facilite o uso do modelo. Muitos dos aspectos de implementação dos *EDs* e *BFs* já foram detalhados ao longo do texto e não necessitam de uma ênfase maior.

Neste capítulo será descrito o Grafo de Controle Adaptativo e o seu processo de síntese, baseado nas informações definidas pelo projetista. Além disso, alguns detalhes relevantes da Plataforma de Controle Adaptativo também serão destacados.

8.1 Síntese do Grafo de Controle Adaptativo

Uma questão considerada muito importante para tolerância a falhas e consequentemente para todo o modelo desenvolvido, é realizar as reconfiguração de maneira eficiente, ou seja, rápida, independente do evento ou da condição que a ativa. A possibilidade da existência de muitas configurações é um fator relacionado diretamente com a eficiência, o que exige soluções apropriadas para este problema. A solução encontrada foi concentrar em um grafo as informações necessárias as reconfigurações. Cada nodo deste grafo corresponde a uma possível configuração ou adaptação do sistema. As arestas que interligam os nodos correspondem as possíveis transições entre

Atributo	**Descrição**
Missão	Identificador da missão associada ao nodo.
Fase	Identificador da fase da missão associada ao nodo.
ID_CONF	Identificador de uma configuração específica dos *PCs*.
ID_SCHED	Identificador de um escalonamento específico de *BFs*.
ID_PROFIT	Identificador da função de ganho.
ID_CONF	Identificador da função de cálculo da confiança.
ID_PERF	Identificador da função de cálculo de desempenho.
Atributos gerais	Identificador de um conjunto de atributos associados à fase como limites de tempo, usos de recursos ou outras informações pertinentes ao controle do robô.
Atributos específicos	Conjunto de atributos específicos da configuração como os tempos de execução esperados, uso de memória e outros.

Tabela 8.1: Dados estáticos existentes em um nodo do *GCA*.

Atributo	**Descrição**
Origem	Identificador do nodo de origem.
Destino	Identificador do nodo de destino.
Finalidade	Identificador com a finalidade da aresta no grafo. Neste caso, corresponde a origem da aresta no modelo desenvolvido.
Atributos específicos	Conjunto de atributos da aresta específicos da finalidade desta.

Tabela 8.2: Atributos comuns a todas as arestas do *GCA*.

as configurações.

O grafo que agrupa as diversas informações necessárias ao controle de alto nível foi chamado de "Grafo de Controle Adaptativo (*GCA*)". Essencialmente, este possui as informações relevantes para todas as trocas de estado do controle, seja por alterações de fase normais, ou por qualquer outro evento. O seu uso simplifica o problema global de decisão a uma pesquisa em grafo, obedecendo determinadas restrições. As arestas são agrupadas em função dos eventos que ativam as transições, de forma a otimizar o processo de adaptação ou reconfiguração.

O grafo *GCA* ($G^{\mathrm{GCA}} = (\eta, \alpha)$) é definido como um conjunto de nodos e arestas direcionadas. Cada nodo $n \in \eta$ representa uma configuração diferente do sistema e cada elemento $e \in \alpha$ representa uma aresta. Os atributos principais de um nodo são apresentados na Tabela 8.1, e os atributos de uma aresta são mostrados nas Tabela 8.2 e na Tabela 8.3.

Finalidade **Atributos**	**Descrição**
Transição de fase	Uma transição de fase normal definida pelo projetista.
Condição	Condição que ativa a transição de fase.
Prioridade	A prioridade da transição em relação às outras.
Recuperação indireta	Uma transição para uma fase equivalente que deve ocorrer quando o índice de sucesso da fase corrente não está satisfatório (Seção 7.1.3).
Prioridade	Prioridade da transição em relação às outras.
Recuperação Geral	Uma transição para uma fase de recuperação (Seção 7.1.4). Utilizada quando é detectada uma falha não diagnosticada ou uma situação imprevista, ou não há tempo para a recuperação específica.
Adaptação	Transição entre configurações diferentes da mesma fase.
Ganho	Indicador da natureza do ganho provável da transição. Opções:(CONF/DESEMP). Este atributo pode ser utilizado para acelerar processos de adaptação, quando os objetivos correntes são alterados
Custo	Custo associado com a adaptação. Embora seja um fator importante, este valor não foi tratado neste trabalho como explicado na Seção 7.2.4.
Recuperação de falha	Transição entre configurações diferentes da mesma fase. Conecta a uma configuração que é provável de tolerar uma nova falha detectada (Seção 7.1.4).
Overhead	Tempo estimado para se completar o processamento do ciclo corrente se efetuando a reconfiguração. Se este tempo já consumido no fluxo, adicionado a este *overhead*, ultrapassar o limite da fase, esta transição não é efetuada. Neste caso, a recuperação de falhas é efetuada com a transição para o nodo apontado pela a aresta com finalidade de "Recuperação Geral".
BFs de teste	Este campo associa os *BFs* que efetuam testes na configuração corrente, com as arestas de recuperação específicas. Esta informação é importante para selecionar a configuração de recuperação adequada.
Identificação da falha	Este campo contem identificadores de *BFs* e *EDs* que podem ter gerado ou propagado a falha que foi detectada. Esta informação é comparada com o resultado do teste que o *BF* que detectou a falha transfere para a *PCA* e para o sistema de diagnóstico.

Tabela 8.3: Atributos de uma aresta do *GCA* de acordo com a sua finalidade.

O processo de síntese deste grafo é composto de várias etapas que se complementam. As etapas principais enumeradas a seguir são detalhadas nas próximas seções.

1. Um único grafo é criado contendo todas as transições de fase das missões e integrando as fases de recuperação e arestas relativas as fases equivalentes.

   Unificação dos autômatos das missões.

   Inserção das arestas de equivalência.

   Inserção das arestas de recuperação.

2. Cada uma das fases é expandida para as suas possíveis configurações. As transições de fase devem ser ajustadas ao novo grafo.

   Inclusão das arestas de adaptação.

   Inclusão das arestas de recuperação de falhas.

Agregação das transições de fase

A primeira etapa do processo é o mapeamento do autômato ($M^{\text{robô}} = (Q, \Sigma, \delta, q_0, F)$) definido na Seção 7.4 para o grafo auxiliar chamado de ($G^{\text{GCA}}_{\text{aux}} = (\mu, \nu)$). A representação de um autômato finito na forma de um grafo direcionado é extremamente usual. Os nodos do grafo são associados diretamente aos estados do autômato ($Q \longrightarrow \mu$), e as arestas direcionadas correspondem às transições juntamente com as condições de ativação, definidas por Σ ($(\delta, \Sigma) \longrightarrow \nu$). Optando-se pelo uso de um grafo, os nodos e arestas devem conter o mesmo conjunto de informações contidas na definição do autômato.

$$\exists a_{i,j}(\text{Transição de fase}, \text{Cond}_c) \in \nu \Longleftrightarrow \exists e_{i,j}(\text{Cond}_c) \in \delta \wedge c \in \Sigma$$

A segunda etapa para composição do *GCA* é adicionar a informação fornecida pelo projetista sobre as fases equivalentes, definidas na Seção 7.1.3. Essencialmente são inseridas duas arestas para cada par de fases equivalentes.

$$\exists a_{i,j}(\text{Recuperação indireta}, \text{Prioridade}_{i,j}) \in \nu \iff$$
$$\text{Fase}_i \equiv \text{Fase}_j \wedge \nexists \text{Fase}_{i,j}^{\text{Preparação}}$$
$$\exists a_{i,k}(\text{Recuperação indireta}, \text{Prioridade}_{i,j}) \in \nu \iff$$
$$\text{Fase}_i \equiv \text{Fase}_j \wedge \exists \text{Fase}_{i,j}^{\text{Preparação}} = Fase_k$$
$$\exists a_{j,i}(\text{Recuperação indireta}, \text{Prioridade}_{j,i}) \in \nu \iff$$
$$\text{Fase}_i \equiv \text{Fase}_j \wedge \nexists \text{Fase}_{j,i}^{\text{Preparação}}$$
$$\exists a_{j,k}(\text{Recuperação indireta}, \text{Prioridade}_{j,i}) \in \nu \iff$$
$$\text{Fase}_i \equiv \text{Fase}_j \wedge \exists \text{Fase}_{j,i}^{\text{Preparação}} = \text{Fase}_k$$

A terceira etapa para composição do *GCA* é adicionar a informação fornecida pelo projetista sobre as fases de recuperação, definidas na Seção 7.1.4. Neste caso são inseridas arestas conectando cada fase da missão a uma fase de recuperação.

$$\exists a_{i,\text{Rec}_i}(\text{Recuperação Geral}) \in \nu \forall i \in Q_m \iff \exists \text{Fase}_i^{\text{Rec}}$$
$$\exists a_{i,\text{Rec}_m}(\text{Recuperação Geral}) \in \nu \forall i \in Q_m \iff \neg \exists \text{Fase}_i^{\text{Rec}}$$

Além disso, as fases de recuperação específicas das missões devem ser conectadas a fase de Recuperação Global.

$$\exists a_{\text{Rec}_m,\text{Rec}_{\text{system}}}(\text{Recuperação Geral}) \in \nu \forall \text{Rec}_m \in Q_{\text{robô}}$$
$$\exists a_{\text{Rec}_i,\text{Rec}_{\text{system}}}(\text{Recuperação Geral}) \in \nu \forall \text{Rec}_i \in Q_{\text{robô}}$$

Após esta etapa, o grafo (GCA) contém todas as fases e transições de fase definidas explicitamente ou implicitamente pelo projetista.

Inclusão das adaptações

O grafo criado ($G_{\text{aux}}^{\text{GCA}} = (\mu, \nu)$) já contem as fases e suas transições, incluindo as informações das fases equivalentes e fases de recuperação. Falta incluir as informações para as adaptações e recuperação de falhas direta. Este processo é mais complexo, pois envolve a expansão de cada fase em suas múltiplas configurações. O maior problema é selecionar a melhor configuração quando se troca de fase.

Quando o sistema realiza uma adaptação pode estar se ajustando a uma alteração decorrente da mudança do ambiente ou da probabilidade de falha de seus elementos. Se uma transição de fase é realizada para uma configuração que não seja adequada, poderá ocasionar problemas no funcionamento, mesmo que depois de alguns ciclos o sistema alcance um estado ideal novamente. Por isso, a informação sobre as adaptações realizadas não pode ser perdida em uma transição de fase.

Para garantir a eficiência nas transições entre fases, é necessário que as configurações entre duas fases conectadas estejam associadas, de forma que as otimizações realizadas em cada fase não sejam perdidas em uma transição. Embora existam várias opções de implementação, a informação de associação deve ser calculada previamente a execução e armazenada em alguma estrutura de dados. A opção selecionada no modelo foi expandir os nodos de fase do grafo de forma a agregar esta informação, explicando em outras palavras, cada fase definida pelo projetista é substituída por um conjunto de nodos, cada um correspondendo a uma configuração diferente.

As configurações de fases de diferentes devem ser associadas de acordo com o seguinte critério. Se uma configuração está com um ganho ótimo na fase corrente para um determinado estado global do sistema, ela deve estar conectada a configurações de outras fases que ofereçam também um ganho ótimo para o mesmo estado global. Esta condição é muito difícil de ser garantida devido à diversidade de opções para o estado global que inclui o estado de falhas do sistema. A solução ótima pode exigir em alguns casos, o cálculo do ganho no momento da transição de fase, o que acarretaria em uma perda de desempenho. Neste caso, optamos por uma abordagem heurística na seleção da melhor associação entre duas adaptações de fases. A heurística de seleção da melhor associação entre a adaptação $C_x^i \in N_i$ da Fase_i com a fase distinta Fase_j foi chamada de β e é mostrada na Equação 8.1.1. Sempre vai existir um resultado único para β, pois a heurística é uma seqüência de critérios de seleção os quais são descritos a seguir:

1. O fator mais importante é que as transições de fase devem manter a confiabilidade do sistema e respeitar recuperações de falhas já realizadas. Portanto, devem possuir os mesmos caminhos redundantes para processar da mesma forma no fluxo as mesmas informações. Como os caminhos são diretamente definidos pelo uso dos *BFs*, quanto maior for o número de caminhos iguais entre duas configurações de fases diferentes, maior será a interseção entre os respectivos conjuntos de *BFs*. Assim sendo, o primeiro critério para β é o tamanho do conjunto interseção entre duas configurações.

   $$\beta(C_x^i, \text{Fase}_j) = C_y^j \iff (C_x^i \cap C_y^j) \geq (C_x^i \cap C_k^j) \forall k \in N_j | k \neq y$$

2. O segundo fator é o desempenho, neste caso, deve-se conectar configurações com a diferença de desempenho previsto (I_D) a menor possível (Seção 7.2.2).

$$\beta(C_x^i, \text{Fase}_j) = C_y^j \Longleftrightarrow ABS(I^D_{C_x^i} - I^D_{C_y^j}) \leq ABS(I^D_{C_x^i} - I^D_{C_k^j}) \forall k \in N_j | k \neq y$$

3. O último critério de seleção é a menor diferença no tempo de processamento do fluxo entre as duas configurações.

$$\beta(C_x^i, \text{Fase}_j) = C_y^j \Longleftrightarrow ABS(T_{C_x^i} - T_{C_y^j}) \leq ABS(T_{C_x^i} - T_{C_k^j}) \forall k \in N_j | k \neq y$$

$$\begin{aligned} & \beta(C_x^i, \text{Fase}_j) = C_y^j \\ & \Longleftrightarrow \\ & (i \neq j \wedge C_y^j \in N_j \wedge C_x^i \in N_i) \wedge (\\ & ((C_x^i \cap C_y^j) \geq (C_x^i \cap C_k^j)) \wedge \\ & (ABS(I^D_{C_x^i} - I^D_{C_y^j}) \leq ABS(I^D_{C_x^i} - I^D_{C_k^j})) \wedge \\ & (ABS(T_{C_x^i} - T_{C_y^j}) \leq ABS(T_{C_x^i} - T_{C_k^j})) \\ &) \forall k \in N_j | k \neq y \end{aligned} \tag{8.1.1}$$

Após a definição de β, o processo de síntese do Grafo de Controle Adaptativo (GCA) pode ser realizado utilizando informações seguintes já processadas em outras etapas.

- O grafo auxiliar $G^{\text{GCA}}_{\text{aux}} = (\mu, \nu)$ já criado.
- Os conjuntos de adaptações ($N_f = (C_1^f, ..., C_n^f)$) geradas para cada Fase_f.
- O conjunto de grafos contendo as informações de adaptação $G_f^A = (N_f, \epsilon_f)$ de cada fase.
- O conjunto de grafos contendo as informações de recuperação direta de falhas $G_f^R = (N_f, \phi_f)$ de cada fase.

A seqüência de passos para definir o $G^{\text{GCA}} = (\eta, \alpha)$ é a seguinte:

1. Um nodo deve ser inserido em α para cada adaptação existente em N_f de todas as fases do autômato final do robô.

$$\exists C_i^f \in \eta \Longleftrightarrow \exists C_i^f \in N_f \forall f \in M^{\text{robô}}$$

2. As arestas de adaptação também devem ser inseridas a partir das informações existentes em cada grafo G_f^A de uma fase. Para cada aresta no grafo em G^A são inseridas duas em G^{GCA} correspondendo ao ganho previsto de confiança e ao ganho previsto de desempenho.

   $(\exists a_{C_i^f,C_j^f}(\text{Adaptação, CONF}) \in \alpha) \wedge (\exists a_{C_j^f,C_i^f}(\text{Adaptação, DESEMP}) \in \alpha) \forall \in a_{C_i^f,C_j^f} \in \epsilon_f \forall f \in M^{\text{robô}}$

3. As arestas de recuperação de falhas também devem ser inseridas a partir das informações existentes em cada grafo G_f^R de uma fase.

   $(\exists a_{C_i^f,C_j^f}(\text{Recuperação de falha}, T^{\text{overhead}}_{C_i^f,C_j^f}, (BFs^{\text{teste}}_{C_i^f})^+, (BFs \vee EDs)^*_{C_i^f}) \in \alpha) \forall \in a_{C_i^f,C_j^f}((BFs^{\text{teste}}_{C_i^f})^+, (BFs \vee EDs)^*_{C_i^f}) \in \phi_f \forall f \in M^{\text{robô}}$

4. As arestas de transição de fase também devem ser inseridas. Cada aresta saindo de uma fase gera uma aresta saindo de cada uma das adaptações da fase.

   $\exists a_{C_i^f,C_j^g}(\text{Atributos}_{f,g}) \in \alpha \iff (\beta(C_i^f, \text{Fase}_g) = C_j^g)) \forall C_i^f \in N_f \forall a_{f,g}(\text{Atributos}_{f,g}) \in \nu$

Após estas etapas, o grafo G^{GCA} contêm todas as adaptações e arestas geradas. Infelizmente, ainda pode existir uma perda de informação quando se realizam transições entre fases com números diferentes de adaptações. Este caso é mostrado na Figura 8.1, no qual existe um ciclo entre dois nodos com números distintos de configurações. Após cada execução do ciclo é necessário repetir o processo de adaptação. A solução escolhida para resolver este problema é multiplicar os nodos e arestas no *GCA* para manter na própria navegação do grafo a informação necessária.

Vale lembrar que as fases associadas a missão do sistema (M^{system}) não possuem múltiplas adaptações e também permanecem únicas durante este processo. O processo de duplicação dos nodos é extremamente simples. A idéia básica do processo é identificar quando uma informação de adaptação foi perdida em uma seqüência de transições, e inserir novos nodos e arestas de forma a preservar a mesma.

1. Primeiramente se identifica um nodo que não recebe arestas de transição de fase existentes em $G^{\text{GCA}}_{\text{aux}}$.

   $\nexists a_{C_i^f,C_j^g} \in \alpha \wedge a_{f,g} \in \nu$

2. O segundo passo é verificar se existe na seqüência de fases que levam a Fase_g a uma adaptação (C_x^k) na qual o seu β em relação a g é a adaptação C_j^g. Nesta seqüência não podem ser consideradas as fases da missão do sistema que não possuem adaptações, e conectam indiretamente todas as missões definidas pelo projetista.

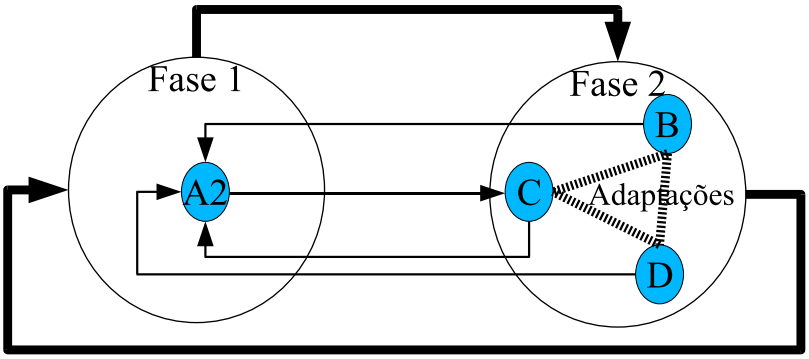


Figura 8.1: Exemplo de perda da informação de adaptação na transição da Fase_2 para a Fase_1.

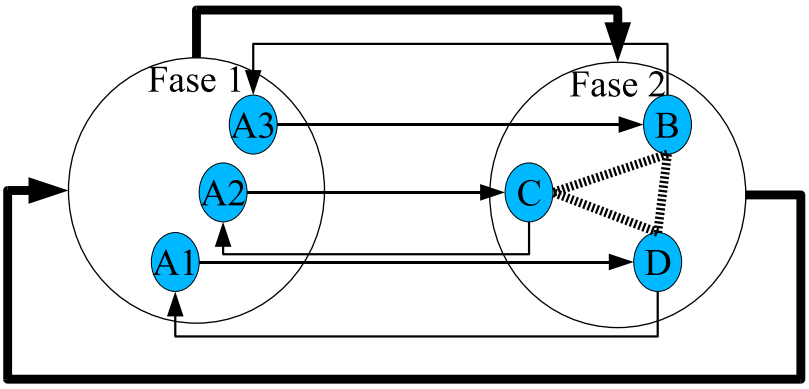


Figura 8.2: Solução para manter a informação de adaptação em transições de fase.

$(\exists C_x^k \in N_k | \beta(C_x^k, Fase_g) = C_j^g) \wedge (\exists \text{Fase}_k \Longrightarrow^* \text{Fase}_g)$

$(\text{Fase}_k \Longrightarrow^* \text{Fase}_g) \Longleftrightarrow (\exists (a_{k,y^1}, a_{y^1,y^2}, ...a_{y^m,g}) \in \nu | m \geq 1 \wedge (k, y^1, ...y^m) \neg \in Q^{\text{system}}$

3. Se for encontrada uma configuração C_x^k que atenda aos requisitos, os nodos participantes da seqüência de C_x^k até a Fase_g devem ser duplicados, juntamente com suas arestas. A adaptação pertencente ao caminho analisado correspondente à fase anterior a Fase_g deve ter sua aresta alterada para se ligar a C_j^g. Para simplificar o exemplo, suponha que a Fase_k se conecte a Fase_g através da Fase_y.

   $(\text{Fase}_k \Longrightarrow^* \text{Fase}_g) \Longleftrightarrow (\exists (a_{k,y}, a_{y,g}) \in \nu | \wedge (k, y) \ni Q^{\text{system}}$

   $\exists a_(C_x^k, C_z^y) \Longrightarrow (\text{Duplicar}(C_z^y) \Rightarrow (\exists C_{z^2}^y \in \eta)) \wedge (a_(C_x^k, C_z^y) \Rightarrow a_(C_x^k, C_{z^2}^y)) \wedge (\exists a_(C_{z^2}^y, C_j^g) \in \alpha) \wedge (a_(C_{z^2}^y, C_b^a) \forall a_(C_z^y, C_b^a) \in \alpha)$

4. O processo deve ser repetido interativamente até que não seja mais possível inserir novos nodos e arestas no grafo. O tamanho da seqüência de fases consideradas no algoritmo deve ser crescente. Na primeira interação avaliam-se todas as seqüências com duas transições. Quando não for mais possível inserir nodos, avalia-se com três transições e assim por diante. O processo deve incluir na avaliação *loops*, mas não é necessário ultrapassá-los.

O Grafo de Controle Adaptativo está completo após esta expansão com os nodos duplicados. Este contém todas as informações definidas pelo projetista ou geradas por processos automáticos. O projetista sempre que desejar poderá alterar as informações contidas no grafo para realizar testes e melhorar o controle implementado.

O exemplo da Figura 8.3 mostra três fases interconectadas após a primeira etapa de expansão do grafo e após a inserção de arestas que conectam as fases. As adaptações D e F não recebem arestas de transição de fase. Neste caso, verificam-se as transições $\text{Fase}_1 \rightarrow \text{Fase}_2 \rightarrow \text{Fase}_3$ e detecta-se $\beta(A, \text{Fase}_3) = F$. Neste caso, um novo nodo $C1$ é inserido na Fase_2, como mostrado na Figura 8.4, garantindo a informação de adaptação no grafo ($A \rightarrow C1 \rightarrow F \rightarrow A \rightarrow ...$). Como na Fase_1 não existem mais nodos com múltiplas arestas de saída, o processo é repetido incluindo a própria Fase_3, pois a mesma faz parte do ciclo $\text{Fase}_3 \rightarrow \text{Fase}_1 \rightarrow \text{Fase}_2 \rightarrow \text{Fase}_3$ e detecta-se $\beta(D, \text{Fase}_3) = D$. Neste caso, são inseridos novos nodos na Fase_1 e Fase_2, finalizando assim, o processo de expansão com o grafo da Figura 8.5.

Redução da memória utilizada pelo *GCA*

À primeira vista, a solução empregada para manter a informação sobre adaptações ampliando o número de nodos do *GCA* aumenta muito os requisitos de memória utilizada pelas estruturas de dados. Algumas opções simples de implementação resolvem

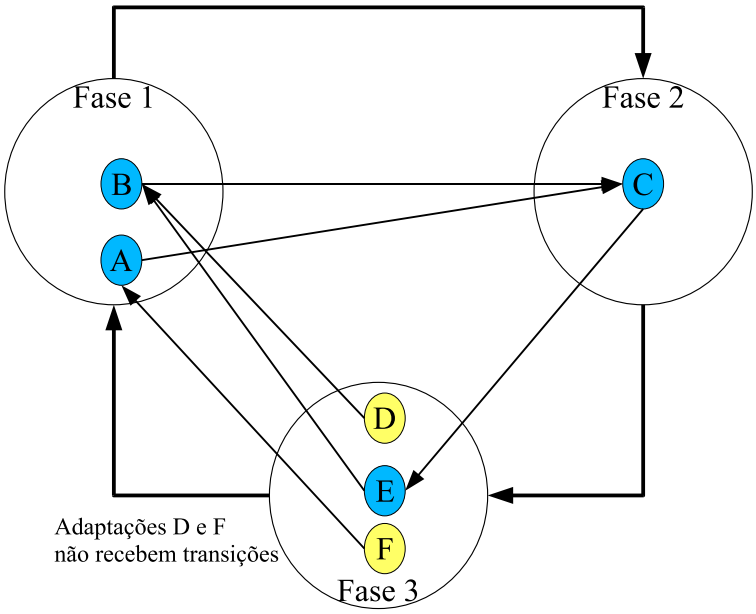


Figura 8.3: Exemplo da duplicação de nodos no grafo para garantir as informações sobre as adaptações.

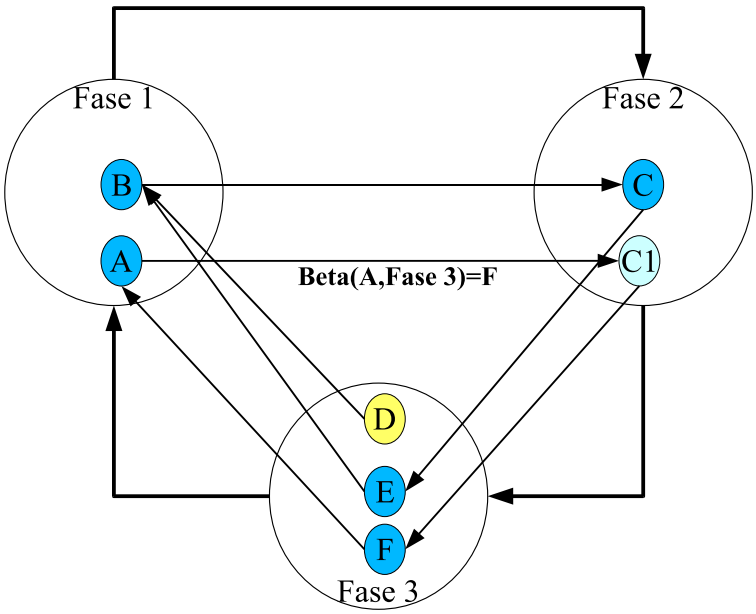


Figura 8.4: Grafo após a duplicação do nodo C para $C1$.

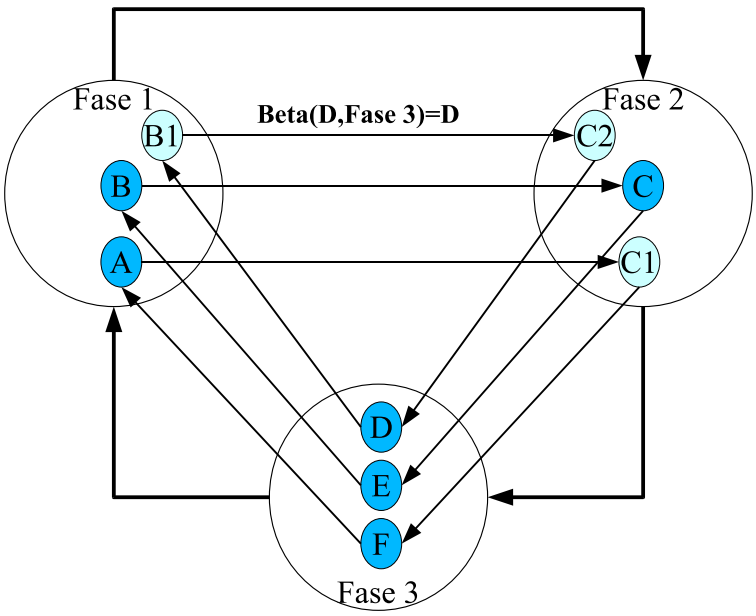


Figura 8.5: Grafo final com todos os nodos necessários inseridos.

Atributo	Descrição
Índice Auxiliar	Índice da estrutura de dados auxiliar que armazena as informações da adaptação correspondente.
Índice das arestas	Índice do conjunto de arestas associadas a este nodo.
Número de repetições	Número desta repetição da mesma adaptação. Esta informação é utilizada para indexar o nodo de destino em cada aresta.

Tabela 8.4: Redefinição de um nodo do *GCA* para reduzir a memória utilizada.

Origem	Vetor de identificadores dos nodos de origem.
Destino	Vetor de identificadores dos nodos de destino.

Tabela 8.5: Redefinição em atributos de uma aresta do *GCA* para reduzir a memória utilizada.

este problema.

Todos os atributos que caracterizam uma adaptação, mostradas na Tabela 8.1, são duplicados na cópia de um nodo. Se for criada uma estrutura extra de dados para armazenar as informações de cada adaptação, é possível compartilhá-la entre todas as cópias dos nodos. Cada elemento desta nova estrutura representa uma adaptação C_i^f, possuindo um índice único.

Além disso, todos os atributos das arestas vão ser iguais entre as várias cópias do mesmo nodo de origem. Se for incluído em cada aresta um vetor de nodos destinos, é possível compartilhar as arestas entre todas as cópias. Para isto ser viável, é necessário indexar este vetor de nodos destinos pelo número da cópia do nodo original. A solução completa fica da seguinte forma:

Os nodos do *GCA*, como mostrado na Tabela 8.4, passam a ter somente três informações: o índice da adaptação na estrutura auxiliar; um índice para suas arestas na tabela de arestas; e seu número de cópia. Desta forma, a informação de cada adaptação pode ser compartilhada por múltiplos nodos, sem impacto significativo na memória utilizada.

A estrutura de arestas é alterada, para conter um vetor de nodos de origem e um vetor de nodos de destino, como pode ser visto Tabela 8.5, mantendo o restante dos atributos inalterados. A indexação do nodo de destino no vetor da aresta pode ser realizada tanto pelo vetor de nodos de origem, quanto pelo número da copia presente nos atributos do nodo de origem.

Com esta e outras soluções simples é possível reduzir muito os requisitos de memória do sistema, principalmente devido ao fato que as adaptações, os escalonamentos e o *GCA* são gerados previamente a execução do controle. Esta característica

permite muitas opções de otimização dos algoritmos utilizados e das estruturas internas de dados.

8.2 Plataforma de controle Adaptativo

A maior parte das informações sobre a Plataforma de Controle Adaptativo foi distribuída ao longo do texto, junto ao detalhamento dos componentes do fluxo e do controle de alto nível. Pode-se dizer, que a plataforma é constituída por um conjunto de elementos.

- As estruturas de dados estáticas correspondendo às informações sobre os *EDs*, *PCs* e *BFs*.
- O Grafo de Controle Adaptativo juntamente com todas as informações associadas:

  os diferentes escalonamentos de *BFs* das adaptações;

  os cálculos de confiança, desempenho e ganho.
- Estruturas de dados dinâmicas que armazenam conjuntos de valores relevantes a cada elemento das estruturas estáticas:

  valores de dados;

  valores de confiança e probabilidades associadas;

  resultados de testes e eventos relacionados a defeitos e falhas.
- Um conjunto de funções disponíveis na *API* para manipular as informações armazenadas.
- Um conjunto de *BFs* especiais incluídos nos escalonamentos para realizar as funções de controle. Um controle de sincronismo com os processos de tempo real. Um escalonador que executa efetivamente os *BFs* e trata os eventos de falhas.

A maior parte destas informações e problemas já foram abordados ao longo do texto, restando apenas discutir a interação do processamento do fluxo com o controle de alto nível. Como já foi dito, o intuito de todo modelo foi de concentrar o processamento no fluxo, criando uma estrutura única a qual fica mais simples ser analisada e avaliada. Existe no conceito do modelo uma hierarquia de controle com dois níveis distintos, mas na implementação real quase todo o processamento é concentrado no fluxo. A execução de cada ciclo do fluxo passa por uma seqüência previamente determinada como mostrado na Figura 8.6 e detalhada a seguir.

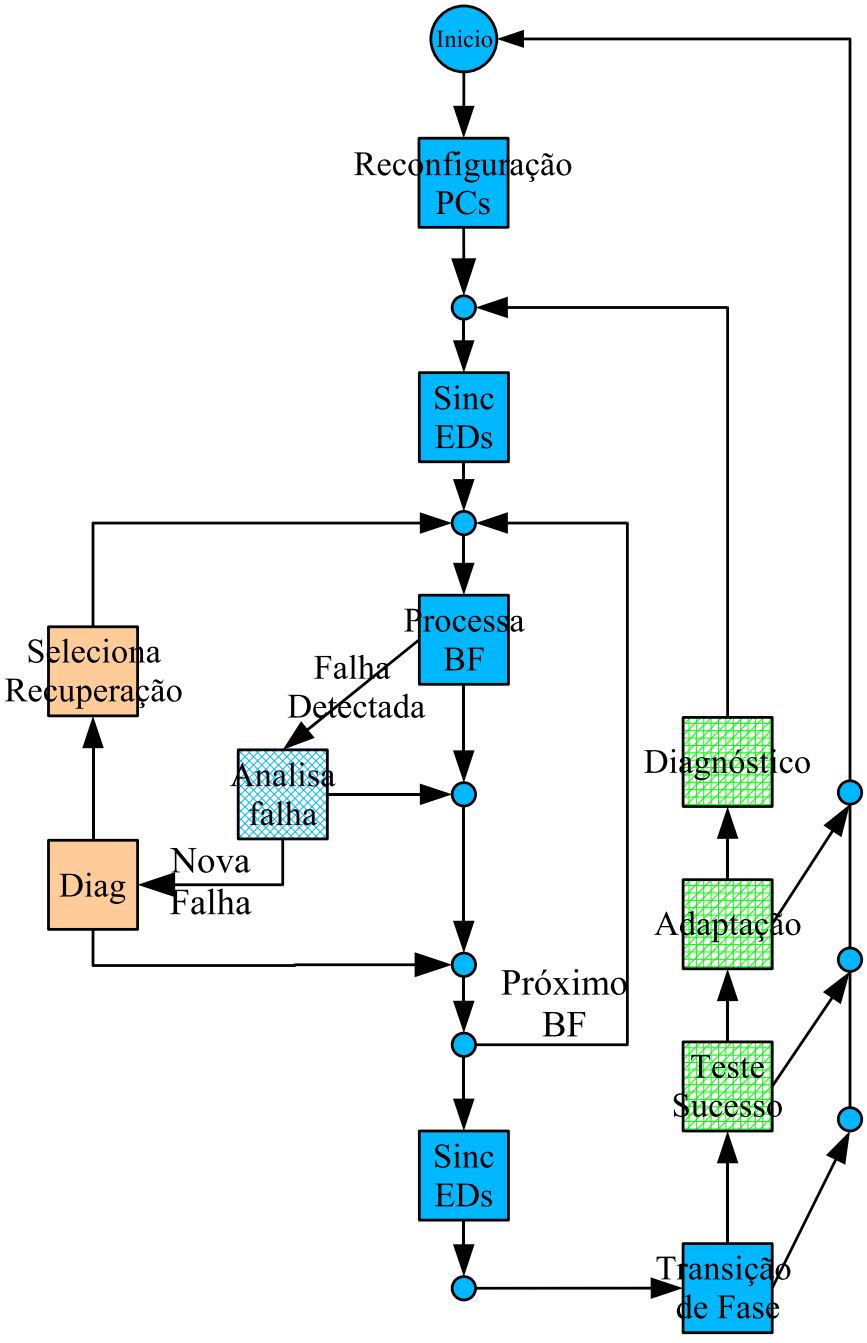


Figura 8.6: Seqüência de processamento no escalonador do fluxo.

1. Caso o nodo do *GCA* corrente tenha sido alterado, o índice de configuração dos Parâmetros de Controle é atualizado. Também, é realizado qualquer outro ajuste interno que se faça necessário.

2. O sincronismo com o processamento de tempo-real é realizado, desta forma o fluxo recebe os valores atualizados para os sensores.

3. Cada um dos Blocos Funcionais é executado.

   O *BF* pode pedir a *PCA* informações adicionais armazenadas junto ao escalonamento. Entre as informações, possíveis estão os atributos definidos nas descrições do modelo e identificadores de outros elementos associados, como *EDs*, *PCs* e outros *BFs*.

   A cada *ED* acessado internamente para leitura, sua confiança é incorporada à confiança das saídas do *BF*.

   Quando um *PC* indexado é acessado pela primeira vez no ciclo, de acordo com as suas propriedades, a validade da sua indexação atual é verificada e se necessário é refeita.

   O *BF* caso faça algum tipo de teste pode retornar um código de erro para a *PCA*.

4. A *PCA* testa o resultado da execução do *BF*. Caso o *BF* detecte uma falha que necessite recuperação, a seqüência é interrompida. Senão, o próximo *BF* da seqüência é selecionado e executado. A seqüência de recuperação de falhas é detalhada após a descrição completa da seqüência normal.

5. O sincronismo com o processamento de tempo-real é realizado atualizando os atuadores.

6. Os testes de transição de fase são efetuados em função da prioridade das arestas. Caso uma transição seja ativada, a *PCA* é informada e a seqüência interrompida.

7. Caso exista arestas conectando o nodo a fases equivalentes, o índice de sucesso é avaliado. Se este estiver inadequado aos requisitos definidos, é realizada uma pesquisa no *GCA* por uma adaptação viável de uma fase equivalente. Caso seja for encontrada, a *PCA* é informada e a seqüência interrompida.

8. O controle adaptativo tenta encontrar uma adaptação com melhor ganho que a configuração atual. Se for encontrada, a *PCA* é informada e a seqüência interrompida. O processo adaptativo pode levar vários ciclos avaliando as alternativas de reconfiguração, mantendo internamente o seu contexto.

9. O sistema de diagnóstico é chamado para atualizar as suas informações.

A seqüência de execução de *BFs* corresponde a um escalonamento específico, o que facilita o controle apurado das ações executadas em cada adaptação individualmente. Por exemplo, quando não se deseja realizar a recuperação indireta, simplesmente não se insere o *BF* especial que controla esta função no escalonamento. Os escalonamentos utilizados no protótipo foram listados no Apêndice E.

Se durante a execução de um escalonamento normal de *BFs* é detectada um falha (item 4), a seqüência de execução é interrompida, e a *PCA* realiza a ação de recuperação adequada, cuja seqüência de passos é descrita a seguir:

1. Se o evento de falha já for conhecido e já foi avaliado pelo sistema de diagnóstico, ele é simplesmente armazenado e a seqüência retorna sua execução normal.

2. Se for um evento novo, a *PCA* procura no *GCA* se existe uma aresta de recuperação de falhas adequada. Como foi visto, a aresta adequada é identificada pelo *BF* que detectou a falha e pelas informações extras que definem a origem da informação incoerente geradas pelo mesmo. Se a aresta existir, e o tempo para executar a adaptação associada for aceitável, a *PCA* altera o nodo corrente e reinicia a execução do novo fluxo saltando os *BFs* já executados na seqüência anterior.

3. Se não for encontrada uma aresta de recuperação específica ou o *overhead* para a recuperação específica é muito grande, a *PCA* altera o nodo corrente para a fase definida pela aresta de recuperação. A execução do novo fluxo é iniciada, se saltando os *BFs* já executados na seqüência anterior.

A ortogonalidade obtida com esta implementação é um fator forte do modelo, pois permite um grande número de configurações de fluxo distintas com finalidades e refinamentos específicos. Estes controlados pela seqüência de funções definidas nos escalonamentos.

Capítulo 9

Protótipo e resultados

> Nossas dúvidas são traidoras e
> nos fazem perder o que, com
> freqüência poderíamos ganhar,
> por simples medo de arriscar.
>
> William Shakespeare
> (1564-1616)

Quando se avaliam sistemas tolerantes a falhas, devem-se comparar os modelos de falhas, juntamente com os tempos de detecção e recuperação, para calcular a disponibilidade e confiabilidade final do sistema. Além disso, é necessário se avaliar a degradação do sistema à medida que os defeitos se acumulam.

Em robôs, quando se comparam diferentes abordagens de controle é comum se escolher uma aplicação simples, para facilitar o processo de teste. Nestes casos, são comparados a taxa de sucesso na execução da tarefa ou missão, o tempo gasto e os recursos consumidos.

Mas neste trabalho de tese, o que se deve utilizado para realizar as comparações? Esta é uma pergunta simples, mas a resposta pode não ser. Isto porque se deve considerar qual é sua contribuição real. A metodologia e o modelo desenvolvidos não correspondem a novas abordagens de controle, ao contrário, tentou-se abstrair ao máximo da abordagem de controle do sistema. O trabalho desta tese propõe um modelo de implementação de sistemas com tolerância a falhas adaptativa. O critério ideal para avaliar o trabalho realizado é comparar o processo de desenvolvimento de um sistema completo com tolerância a falhas adaptativa.

> O critério ideal para avaliar o trabalho realizado seria compará-lo com outros processos de desenvolvimento de sistemas com tolerância a falhas adaptativa.

Comparar o processo de implementação tem o seguinte significado: comparar o desenvolvimento de um ou mais projetos diferentes de sistemas tolerantes a falhas adaptativos utilizando a metodologia proposta e outras metodologias diferentes. Existem alguns pontos que dificultam este tipo de avaliação:

- Precisa-se de projetos de desenvolvimento de sistemas com tolerância a falhas adaptativa. Os principais exemplos encontrados na literatura são satélites e naves espaciais, entretanto não são disponíveis os dados detalhados sobre o desenvolvimento. Encontrar um *benchmark* apropriado é uma tarefa difícil.

- Qualquer projeto complexo o suficiente para fornecer dados significativos requer um longo tempo de desenvolvimento. Não existem projetos pequenos neste campo, sendo todos de grande porte e grande investimento.

- A validação dos resultados de metodologias de desenvolvimento requer que os testes sejam feitos com várias equipes distintas, e se possível com projetos diferentes.

Devido a estes fatores, a analise foi simplificada para permitir a comparação da metodologia e obtenção de resultados. Um protótipo do controle de um robô foi desenvolvido para validar a eficácia da metodologia e fornecer dados experimentais sobre o controle adaptativo. A comparação do uso da metodologia de desenvolvimento de sistemas com tolerância a falhas adaptativa em relação aos projetos "ad-hoc"fica como trabalho futuro.

Desenvolver o controle completo de um robô utilizando toda a metodologia também não é um trabalho pequeno. Muitos pontos do trabalho realizado são apropriados para o uso de ferramentas de auxílio automatizadas, se possível embutidas em um ambiente de desenvolvimento específico. A solução encontrada foi criar um protótipo, no qual todas as estruturas de dados foram criadas manualmente. Além disso, com intuito ainda de simplificar o projeto, apenas um conjunto essencial das funcionalidades previstas para a *PCA* foram implementadas. Para reduzir a complexidade das estruturas de dados criadas manualmente, a missão escolhida para ser realizada pelo protótipo foi a mais simples possível.

Os resultados obtidos na sua maioria indicam apenas a capacidade do projetista, no caso o autor desta tese, de implementar um controle simples de um robô com alguns recursos de tolerância a falhas. O resultado mais pertinente referente ao modelo proposto é o *overhead* inserido no protótipo decorrente da utilização do controle adaptativo e da utilização dos *EDs* como método de comunicação. Estes pontos serão mais bem esclarecidos ao longo do capítulo.

Este capítulo é estruturado da seguinte forma. Alguns aspectos do robô e de seu ambiente de desenvolvimento, juntamente com o simulador utilizado são detalhados.

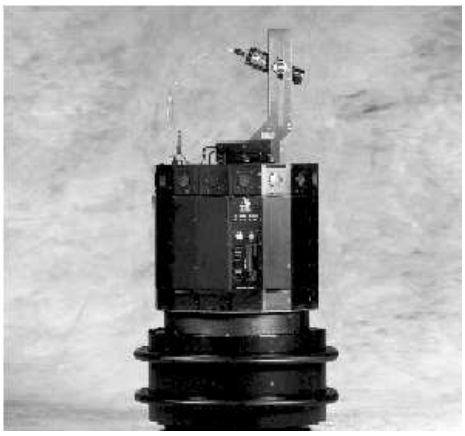


Figura 9.1: Foto do robô *Nomad 200.*

A seguir, a missão é descrita com alguns detalhes de implementação. Por último são detalhados e analisados os resultado conjuntamente com os métodos empregados.

9.1 O robô Nomad 200

O robô utilizado para o desenvolvimento do protótipo foi o *Nomad 200* criado pela empresa *Nomadic Technologies Inc.* A empresa disponibilizava, juntamente com o robô, um conjunto de aplicações apropriadas para o desenvolvimento de aplicações de controle, incluindo um simulador próprio chamado de *Cognus.* Estes pacotes de software associados são descritos no manual do usuário [Nomadic, 1997c] e no manual de referência da linguagem [Nomadic, 1997b]. Devido à simplicidade de uso juntamente com as possibilidades oferecidas por suas características de hardware, o *Nomad 200* foi muito utilizado em pesquisas e no meio acadêmico. Nesta seção serão detalhadas algumas características do robô e do ambiente para melhor compreensão do protótipo.

O *Nomad 200*, mostrado na Figura 9.1, é um robô não-holonomico (*non-holonomic*) com raio de giro zero (*zero gyro-radius*). Possui um conjunto de três rodas as quais podem girar conjuntamente em torno do próprio eixo (*Steer*) e podem movimentar para frente ou para trás (*translation*). A base do robô se conecta com a torre (*turret*) a qual tem a capacidade de girar completamente. As velocidades de rotação em torno do eixo das rodas e da torre variam em intervalos de $0.1\frac{\text{graus}}{\text{segundo}}$ de $-45\frac{\text{graus}}{\text{segundo}}$ até $+45\frac{\text{graus}}{\text{segundo}}$. A velocidade de translação varia em intervalos de $0.1\frac{\text{polegada}}{\text{segundo}}$ de $-240\frac{\text{polegada}}{\text{segundo}}$ até $+240\frac{\text{polegada}}{\text{segundo}}$.

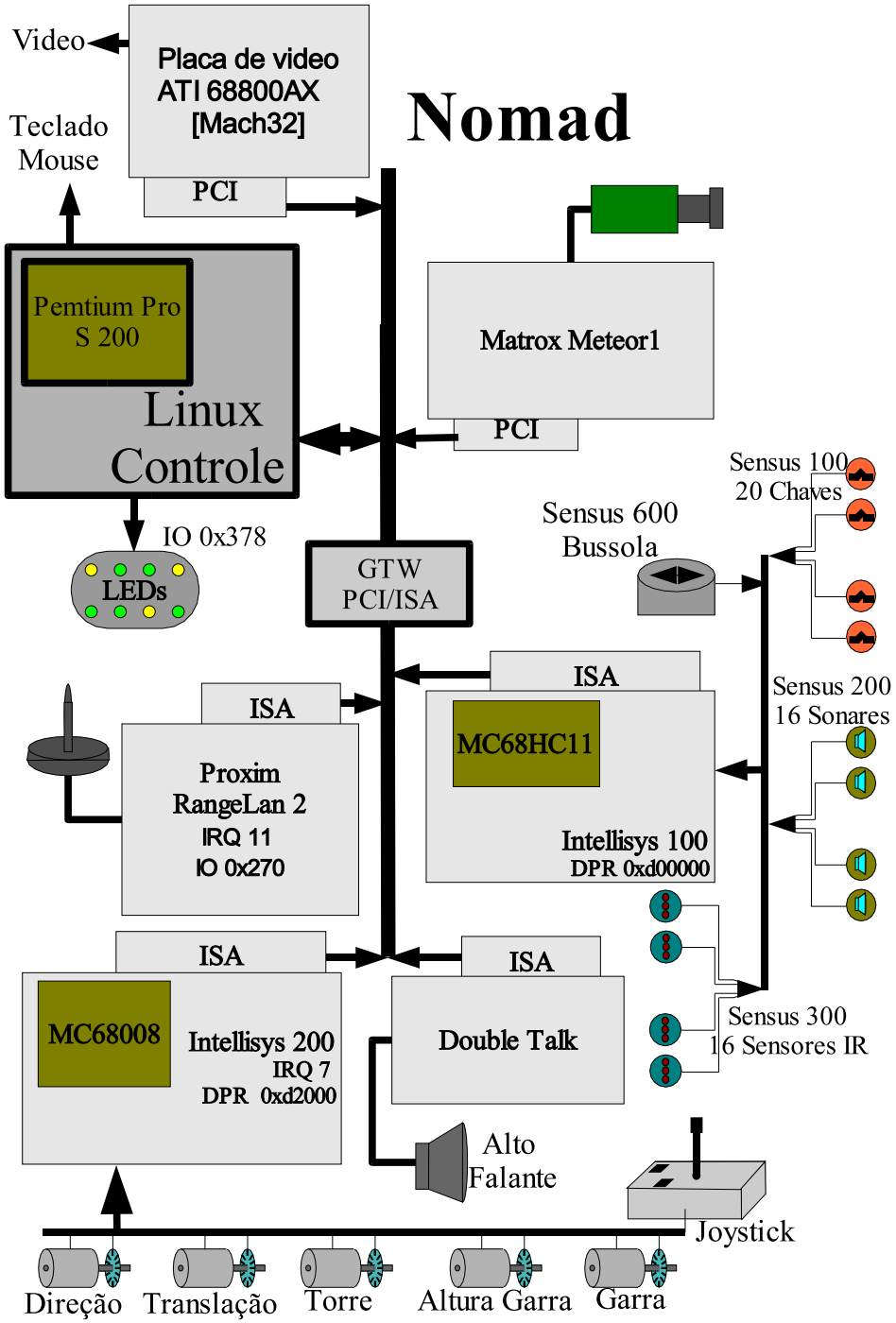


Figura 9.2: *Hardware* de processamento do controle do robô *Nomad 200*.

Na torre do robô se encontra uma garra com movimentação e os sensores relacionados à percepção do ambiente. O robô tem a percepção do ambiente realizada através de sensores de contato, uma câmera de vídeo, uma bússola, sensores infravermelhos e sonares. No protótipo desenvolvido não foram utilizadas a garra, a câmera e a bússola.

Na torre perto da base estão situados dois anéis um anel de borracha com 20 chaves individuais (*bumper sensors*) que indicam simplesmente a realização a realização de contato ou colisão com algum objeto.

Os 16 sensores infravermelhos (IR) fornecem medidas com resolução de 2 polegadas através de valores de [0 a 15], que correspondem às distâncias de obstáculos variando de 0 até 30 polegadas. A medida de valor 15 significa que o obstáculo está a uma distância maior ou igual a 30 polegadas. Os sensores são distribuídos uniformemente ao redor da torre a cada intervalo de $22,5^o$. A abertura do feixe de cada sensor infravermelho corresponde a 10^o graus.

Os 16 sensores de ultra-som ou sonares são distribuídos verticalmente alinhados com os sensores IR, fornecendo valores para as medidas de [17 a 255] associados a intervalos de 1 polegada. O valor de 17 corresponde a distâncias de obstáculos variando de 0 até 17 polegadas e o valor de 255 corresponde a distâncias iguais ou maiores de 255 polegadas. A abertura do feixe de cada sonar corresponde a 12.5^o.

O *Nomad 200* possui uma arquitetura de controle multiprocessada, que pode ser vista na Figura 9.2 na Página 186. A tarefa de controle do robô é executada como um processo normal do sistema operacional *Linux*. O *Linux* executa em um processador *Pentium 200* da placa mãe, a qual se comunica através do barramento *ISA* com outras duas placas processadas: a Intellisys 100 e a Intellisys 200.

A placa *Intellisys 100* possui um processador motorola *MC68HC11* e é responsável pelo controle do conjunto sensorial do robô. Incluindo os sonares, os sensores IR, os sensores de colisão e a bússola. Esta placa executa internamente um software de tempo-real que configura e coleta os dados dos sensores.

A placa *Intellisys 200* possui um processador motorola *MC68008* sendo responsável por controlar todos os motores e os sensores específicos associados a eles. Esta placa também executa internamente um software próprio de tempo-real que além de controlar os motores, coletar os dados associados à velocidade, deslocamento ou posição de cada um.

O Linux utilizado na placa mãe é um *Red Hat 7.2* comum com um *device driver* extra específico (i200m) para a comunicação com a placa *Intellisys 200*. Este *device driver* é necessário porque a comunicação com a placa envolve tratamento de interrupções.

A arquitetura de *software* do *Nomad200* é mostrada na Figura 9.3. O *daemon robotd* é uma tarefa que executa no *Linux* da placa mãe. Este processo se comunica

com a *Intellisys 100* através de mapeamento de memória do barramento *ISA* (*dual-port ram*) e com a *Intellisys 200* através do *device driver* i200m. A tarefa *robotd*, além de ser acessível por conexões TCPIP, é capaz de compartilhar um vetor de valores que representa o estado do sistema e receber um conjunto de comandos de configuração e de comandos de atuação.

Quando é desenvolvido um controle para o *Nomad 200*, o projetista tem disponível uma *API* específica com acesso ao vetor de estados e acesso aos comandos disponíveis, descritos em [Nomadic, 1997b]. Existem duas bibliotecas com esta mesma *API* disponível, uma conectando diretamente com a tarefa *robotd* do it Nomad 200, e a outra conectando com a tarefa *Nserver* que pode estar executando em algum *Unix* e corresponde ao simulador *Cognus*. Esta forma possibilita ao projetista configurar seu programa para utilizar o robô real ou simulador apenas trocando a biblioteca utilizada.

O *Cognos* ([Nomadic, 1997c]) é um simulador específico do *Nomad 200* e é capaz de receber os mesmos comandos e configurações disponíveis no *robotd*. Oferece também uma interface gráfica capaz de apresentar um ou mais robôs em deslocamento em um ambiente virtual com acesso as informações sensoriais básicas destes. A interface do *Cognos* é apresentada na Figura 9.4. No ambiente virtual é possível inserir um conjunto de obstáculos compondo um mapa do ambiente. O projetista é capaz de criar vários mapas distintos do ambiente para testar o seu controle e realizar experimentos sem o risco de danificar o robô real. Os dados numéricos coletados nos experimentos realizados nesta tese foram obtidos com o uso do *Cognos*.

Todos os elementos de software do *Nomad* foram encontrados na Internet e executam em um Linux *Red Hat*. Parte do trabalho implementado nesta tese foi realizar a atualização do *device driver* i200m para o *Red Hat 7.2* e a compilação do restante do ambiente no mesmo.

A arquitetura de software do *Nomad* facilitou vários aspectos da implementação do protótipo, entre eles o acesso aos dados sensoriais e controle de baixo nível da atuação. O sincronismo dos *EDs* associados aos sensores foi implementado com a atualização do vetor de estados mantido pelo *robotd*. O sincronismo dos atuadores foi implementado com o agrupamento de todos os comando de atuação em uma mesma função. Em outras palavras, não foi necessário criar funções de tempo real ou cuidar de acesso em áreas críticas para implementar o protótipo.

9.2 Definição da missão

Para a realização dos testes do protótipo foi necessária a definição de uma missão a ser executada. Como visto anteriormente o objetivo do protótipo é validar o modelo proposto. Para tanto, missão deve atender as seguintes premissas básicas:

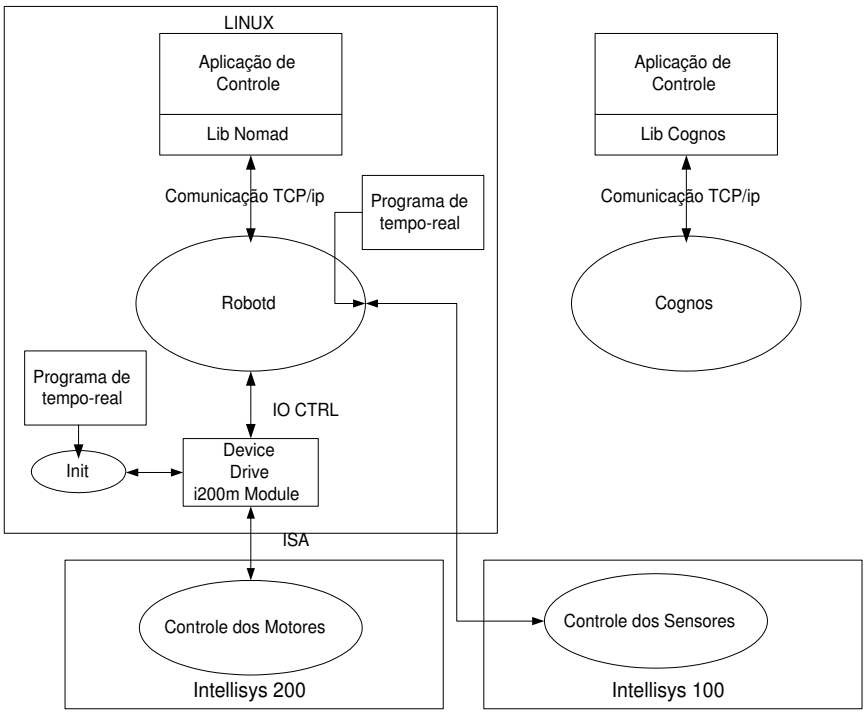


Figura 9.3: Arquitetura de software do *Nomad 200*.

- Ser simples o suficiente para permitir sua a implementação sem o uso de ferramentas de auxilio.
- A execução de uma missão deve ser dividida em fases diferentes.
- Possibilitar a utilização da redundância existente no *Nomad 200*.
- O processo de inserção de falhas deve ser simples e deve permitir a avaliação do impacto destas.
- Deve ser possível criar um conjunto de adaptações ou de políticas de redundância distintas com características de confiabilidade e desempenho diferentes para exercitar as características adaptativas.

A missão selecionada foi movimentar o robô de uma posição inicial até uma posição final sem colidir com obstáculos presentes no ambiente. O desempenho da missão é dado pelo tempo total na realização da mesma. A execução da missão é considerada com sucesso quando o robô chega à posição de destino predeterminada sem colidir

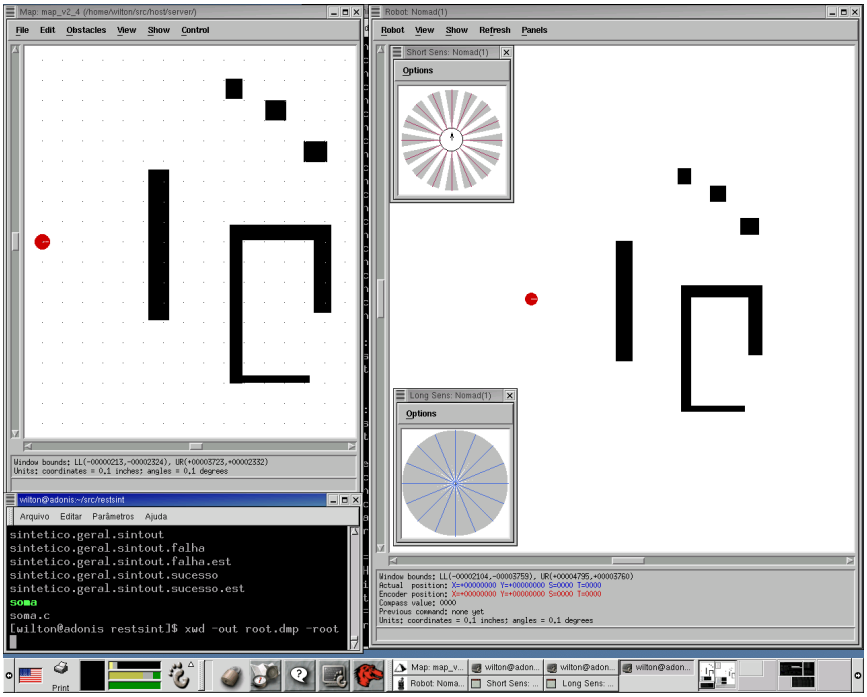


Figura 9.4: Interface gráfica do *Cognos*, que é o simulador do *Nomad 200*.

com nenhum obstáculo em um tempo menor que um limite previamente definido. O tempo limite da missão foi escolhido como sendo o dobro do tempo médio quando não existem falhas.

As distâncias dos obstáculos, provenientes dos sonares e sensores IR, foram utilizadas com fontes de informação redundantes para a implementação da tolerância a falhas. O modelo de falhas que foi definido para o protótipo é composto por leituras totalmente aleatórias nos sensores infravermelhos e nos sonares. Embora as falhas inseridas sejam aleatórias, são sempre geradas dentro da faixa de valores válidos de cada sensor, garantindo assim que a detecção da mesma necessite do uso das informações redundantes provenientes de outros sensores.

A missão foi dividida em um conjunto de fases mostradas na Figura 9.5 e descritas na Tabela 9.1. A execução de uma missão do robô no Cognos é mostrada na Figura 9.6.

Um ponto chave na validação do protótipo é realizar o processamento de informações redundantes existentes. Os sensores IR e sonares do *Nomad* possuem

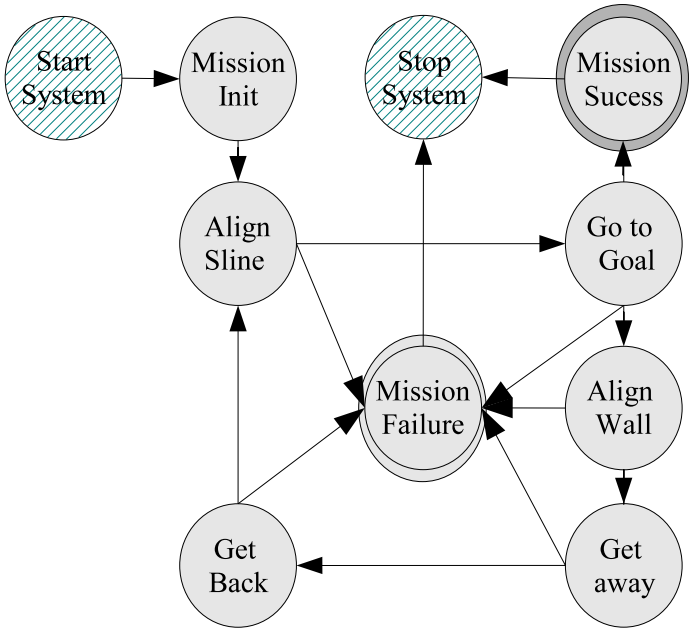


Figura 9.5: Autômato finito que descreve a missão do protótipo.

alcances, resoluções e ângulos de abertura diferentes, sendo alguns considerações destacadas a seguir:

- Devido às diferenças de resolução dos sensores IR e sonares, leituras de distâncias com diferença de até 2 polegadas entre dois sensores alinhados é plenamente aceitável sem ser considerada um indicio de falha.
- Uma leitura de valor de 15 gerada por um sensor IR significa uma distância maior ou igual a 30 polegadas. Este valor é compatível com a leitura de qualquer sonar com valor maior ou igual a 29.
- Uma leitura com o valor de 17 gerado por um sonar significa uma distância menor ou igual a 17 polegadas. Este valor é compatível com leituras de um sensor IR variando de 0 a 9 que correspondem a distâncias variando de 0 a 18 polegadas.
- Para possibilitar a execução da missão utilizando apenas um tipo de sensor ou os dois simultaneamente foi escolhida a distância de 23 polegadas para o robô se manter afastado de um obstáculo. Os sonares oferecem distâncias de $17a255$ e os sensores IR de $0a30$. Removendo os limites imprecisos (17 para os sonares e

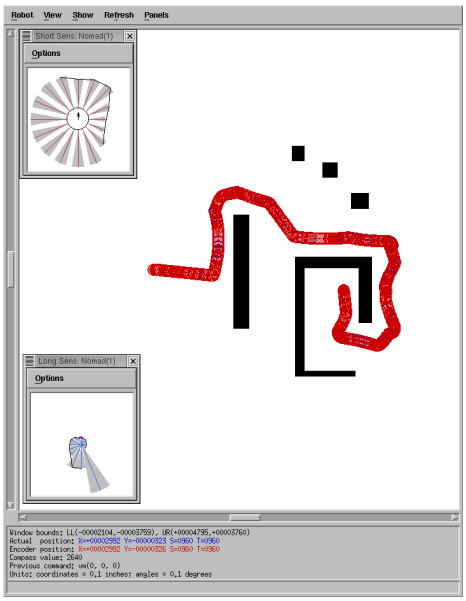


Figura 9.6: Imagem da execução de uma missão do controle utilizando o Cognos.

30 para os sensores IR) temos a faixa de 18 polegadas até 28 polegadas comum aos dois tipos de sensores, A metade desta faixa corresponde ao valor de 23 polegadas ($(18 + (28 - 18)/2) = 23$).

- Como o ângulo de abertura dos sonares e dos sensores IRs é diferente, será comum obter valores incoerentes de sensores com funcionamento corretos quando a medida é feita perto da quina de um obstáculo. Neste caso, um sensor pode detectar o obstáculo enquanto o outro o ignora, gerando medidas incoerentes entre os dois sensores alinhados. Nos experimentos executados foi avaliado um limite de 20% de leituras incoerentes em sensores alinhados com o sistema sem a presença de falhas.

O processo de inserção das falhas no protótipo foi implementado de forma extremamente simples, aproveitando-se da arquitetura disponível. Em cada execução do protótipo, um conjunto de sensores defeituosos é selecionado de forma aleatória. O tamanho do conjunto, ou seja, o número de defeitos é predeterminado para cada execução. As leituras sensoriais do robô são obtidas a partir de um vetor de estados presente nas bibliotecas de software do *Nomad 200*. A inserção de falhas foi implementada no processo de sincronismo com os dados deste vetor através da substituição dos valores corretos por valores aleatórios válidos dentro da faixa válida para o tipo do sensor. Ou seja, para um sensor IR um valor aleatório entre 0 e 15 e para um sonar

Fase	**Descrição**
MissionInit	Conecta com o *robotd* ou com o *Cognos* e inicializa dados internos.
AlignSline	Alinha o robô com o ponto de objetivo.
GoToGoal	Movimenta o robô em linha reta diretamente para o objetivo, até que detecte um obstáculo a frente.
AlignWall	Alinha o robô com um obstáculo encontrado a sua direita.
GetAway	Acompanha o obstáculo a sua direita até que se afaste uma distância determinada do ponto em iniciou a fase.
GetBack	Acompanha o obstáculo a sua direita por um tempo determinado ou até que comece a afastar do objetivo.
Stop	Desativa o controle.
MissionSucess	Cessa a movimentação do robô e contabiliza um missão com sucesso.
MissionFailure	Cessa a movimentação do robô e contabiliza uma missão com falha.

Tabela 9.1: Fases da missão executada no protótipo.

entre 17 e 255. A substituição de uma leitura correta por um valor aleatório pode ser determinada por uma probabilidade de falha associada a cada sensor. Com esta estrutura simples de inserção de falhas foi possível variar tanto o número de defeitos, quanto a gravidade de cada um individualmente no protótipo.

Além das informações de distância instantâneas provenientes dos sensores distintos, é possível aumentar o uso de informações redundantes no protótipo se forem considerados os valores obtidos ao longo do tempo. O uso de informações históricas permite melhorar a qualidade dos valores de distâncias utilizados e facilita a identificação de sensores defeituosos. A solução que se mostrou mais adequada foi à criação de um mapa do ambiente. Neste mapa, poderiam ser combinadas as informações provenientes de todos os sensores, obtendo assim uma confiabilidade maior das informações armazenadas.

O mapa foi implementado como uma matriz (255×255) com resolução de uma polegada com o robô sempre em seu centro. A movimentação do robô implica em uma movimentação dos dados no mapa. Cada posição é inicializada com o valor 0 representando ausência de informação. Um obstáculo é representado por um valor positivo e a ausência de obstáculos ou espaço livre com um valor negativo. Quanto maior for o módulo do valor armazenado, maior a certeza na informação disponível. O mapa é acessível como se fosse um terceiro tipo de sensor de distância, o qual recebe o alinhamento atual dos sensores reais e retorna a distância do obstáculo armazenado.

Além da redundância, o teste de detecção de falhas e o diagnóstico do sistema teve que ser definido. Ambos foram implementados de maneira simples. Uma função

Topologia	Descrição
IR	Utiliza apenas os sensores de proximidade por infravermelho.
IRSN	Utiliza simultaneamente os sonares e os sensores infravermelhos.
IRSNT	Utiliza simultaneamente os sonares e os sensores infravermelhos. Os teste de detecção de falhas nos *EDs* associados às distâncias de obstáculos estão ativos.
IRSNMT	Utiliza simultaneamente os sonares e os sensores infravermelhos e informações de distância, provenientes de um mapa de obstáculos do ambiente construído dinamicamente. Os teste de detecção de falhas nos *EDs* associados às distâncias de obstáculos estão ativos. Esta configuração é mostrada na Figura 7.15 da Página 156.

Tabela 9.2: Configurações de adaptações do protótipo avaliadas.

de teste foi desenvolvida realizando a comparação de dois ou três valores de cada vez. Quando a função avalia dois valores correspondendo a um sensor IR e um sonar e detecta valores incompatíveis, a confiabilidade dos *EDs* associado aos dois sensores é reduzida. Quando a função avalia três valores (um sensor IR, um sonar e uma leitura do mapa equivalente) e detecta valores incompatíveis, as confianças de um ou de ambos *EDs* associados aos sensores reais são reduzidas. Quando são detectados valores coerentes, as confianças são incrementadas. A confiabilidade dos valores lidos do mapa é calculada em função da confiabilidade dos sensores que provêm à informação e da certeza existente no mapa da informação armazenada. A probabilidade de detecção de falhas do teste implementado foi calculada a partir das considerações feitas sobre as leituras dos sensores e da natureza aleatória das falhas.

As possíveis configurações de redundância do protótipo foram exemplificadas no Tabela 7.15 da Página 146. Elas incluem o uso de sensores IR, sonares e o mapa. As seis possíveis configurações do protótipo incluindo a possibilidade dos testes de detecção de falhas foram exemplificadas na Tabela 7.17 da Página 150. Quatro destas apresentadas na Tabela 9.2, foram selecionadas para o facilitar os testes.

O índice de confiança (Seção 7.2.1) de cada adaptação foi implementado com base no cálculo descrito na Seção 6.5. A confiabilidade de todas as fases foi (Seção 6.5.1) considerada equivalente e baseada apenas na confiabilidade dos *EDs* contendo a distância dos obstáculos após junção dos valores redundantes (RD0...RD15).

O índice de desempenho (Seção 7.2.2) de cada configuração também foi considerado igual para todas as fases. Cada configuração diferente foi executada várias vezes e o tempo médio de execução da missão foi normalizado como mostrado na Equação 7.2.1 da Página 138. O índice de ganho (Seção 7.2.2) utilizado em todas as fases e configurações foi definido na Equação 7.2.5 da Página 140, sendo o fator de balanceamento entre confiabilidade e desempenho com valor igual a 0.5 ($F = 0.5$).

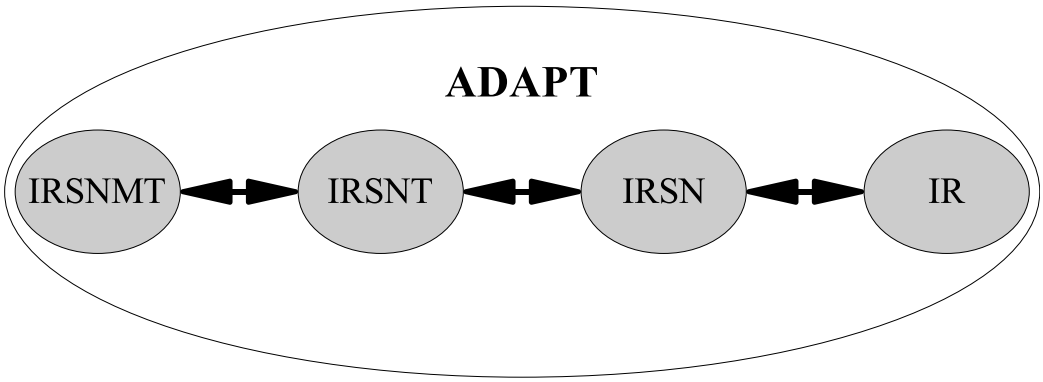


Figura 9.7: Transições de adaptação utilizadas no protótipo.

As transições adaptativas foram definidas conectando IR com IRSN, IRSN com IRSNT, e IRSNT com IRSNMT, como mostrado na Figura 9.7. A execução do controle realizando as adaptações, mostradas na figura, entre estas configurações distintas foi chamada de ADAPT. Para cada um das configurações IR, IRSN, IRSNT foi definido um índice mínimo de confiança, abaixo do qual a configuração não é mais selecionada no processo adaptativo.

As adaptações do protótipo foram consideradas suficientes para se recuperar das falhas, pois a distância mínima mantida dos obstáculos e a velocidade máxima do robô garante um tempo aceitável para se efetuar uma adaptação, que o recupere da falha. Neste caso, para simplificar o controle implementado, não foram incluídas arestas de equivalência (Seção 7.1.3) ou fases de recuperação (Seção 7.1.4) ou arestas de recuperação de falhas (Seção 7.4).

Um conjunto de valores de Parâmetros de Controle (Seção 6.4) foi definido para cada conjunto de adaptações criado. Os parâmetros utilizados no protótipo são mostrados no Apêndice C. Os parâmetros automáticos foram implementados, mas os indexados não. Os testes de transições de fase também foram implementados e são mostrados no Apêndice D. Todos os testes definidos na Seção 7.1.2 foram implementados, mas não houve a necessidade de implementar as equações lógicas entre eles. Os escalonamentos de *BFs* criados manualmente são mostrados no Apêndice E. Finalmente, o Grafo de Controle Adaptativo criado para o protótipo é apresentado no Apêndice F.

A memória ocupada por todas as estruturas de dados, incluindo os arquivos de entradas e estruturas estáticas na linguagem *C*, foi calculada obtendo-se um valor menor do que 5 KBytes. O protótipo possui definido através de estruturas da linguagem *C*: 36 Blocos Funcionais; 106 Elementos de Dados; e 22 Parâmetros de Controle.

Nos arquivos de dados foram definidos: 17 testes de transição; 23 escalonamentos diferentes de *BFs*; 4 configurações automáticas de parâmetros; e o *GCA* possui 26 nodos e 118 arestas.

9.3 Resultados obtidos

A estrutura de acesso aos *EDs* é um ponto chave na capacidade do modelo de facilitar a composição de múltiplas adaptações para uma mesma fase. Por isso, o primeiro ponto que foi avaliado no sistema foi o *overhead* introduzido no fluxo pela utilização das funções de acesso aos *EDs*.

Essencialmente, o cálculo de confiabilidade realizado à medida que os valores são propagados no fluxo (Seção 6.3.2) e a manipulação de múltiplos valores em um *ED* são as funcionalidades básicas utilizadas para facilitar a reconfiguração do fluxo. Caso não fossem necessárias estas funcionalidades, a comunicação entre os *BFs* do fluxo poderia ser realizada utilizando posições de memória predefinidas.

A identificação do tempo de processamento consumido por cada função no protótipo foi realizada com a instrumentação do código através da diretiva "-pg" de compilação do *gcc* do *Linux Red Hat 7.2*. Esta diretiva embute no código gerado a coleta de informações de tempo e freqüência de uso de cada função do programa. Com o aplicativo "*gprof* é possível gerar um arquivo texto com os dados coletados, sendo a saída mostrada no Apêndice G. As funções foram agrupadas por finalidade no sistema, sendo estes agrupamentos detalhados na Tabela 9.3. A porcentagem do tempo consumido por cada agrupamento de funções foi contabilizada e mostrado na Tabela 9.4 e no gráfico da Figura 9.8.

Para melhor compreensão dos dados é importante destacar que o número de ciclos de processamento e duração destes, em cada uma das configurações foi diferente. Os dados foram coletados em função da execução completa da missão. Esta variação é mostrada na Tabela 9.5 e no gráfico da Figura 9.9.

Observando a Tabela 9.4 e o gráfico da Figura 9.8, a primeira questão que se destaca é a variação de processamento proporcionada pelo uso mapa (IRSNMT) entre as diversas configurações. O processamento efetivo que foi consumido internamente pelos *BFs* do fluxo definidos pelo projetista varia de $4,71\%$ até $76,23\%$. Por maior que seja a variação, este resultado é compatível com a variação da complexidade do controle implementado. Pode-se dizer que o controle é um autômato híbrido que possui no baixo nível uma abordagem reativa de pequena complexidade e pouco processamento. A manipulação do mapa que envolve alguns cálculos geométricos para posicionamento e recuperação dos obstáculos consome aproximadamente 27 vezes mais processamento que todo o restante do controle.

Outro detalhe interessante é o crescimento do processamento utilizado pelos *EDs* entre as configurações IR e IRSN devido ao uso de valores redundantes. Este acréscimo

Agrupamento	Descrição
CONTROLE	Funções responsáveis pelo processamento da percepção e atuação.
MAPA	Funções responsáveis pela manipulação do mapa do ambiente criado. Também faz parte do controle, mas foi contabilizado separadamente.
ED	Funções responsáveis pelo acesso e processamento interno aos *EDs*.
ADAPT	Funções responsáveis pelas avaliações das possíveis adaptações de cada fase. Inclui também o calculo do índice de confiabilidade e de ganho.
DIAG	Funções responsáveis pelos testes de detecção de falhas e pelos ajustes da confiabilidade dos sensores.
SCHEDULE	Funções responsáveis pelo controle da execução dos *BFs*.
INIT	Funções responsáveis pela inicialização do sistema e leitura dos arquivos de dados.
SIMULADOR	Funções internas a *API* do *Nomad* responsáveis pela interface com o *Cognos*.

Tabela 9.3: Agrupamentos de funções avaliadas.

reduz o levemente o percentual consumido pelas funções de controle. O mesmo acontece quando o teste de detecção de falhas é introduzido.

O *overhead* de processamento dos *EDs* varia de um máximo de $52,07\%$ na configuração IRSN até o mínimo de $12,48\%$ com o uso do mapa. A variação de processamento dos *EDs* é proporcional aos acessos a eles durante a execução do fluxo, se mostrando totalmente independente do crescimento de processamento interno aos *BFs*. Pode-se dizer que a implementação do modelo se mostra com um *overhead* aceitável quando a complexidade do controle de baixo nível é maior do que a percebida em uma abordagem reativa. Em controles muito simples, o uso de uma estrutura de reconfiguração como a existentes nos *EDs* pode tornar o sistema inviável. Podem-se utilizar outras abordagens na implementação do modelo, que minimizem este *overhead*, entretanto apresentam outros custos associados. Mas esta questão fica com os pensamentos futuros.

As funções de teste, diagnóstico e adaptações também representam um *overhead* substancial nas configurações mais simples. Entretanto, estes recursos essenciais para a implementação de tolerância a falhas adaptativa demandam processamento independente da abordagem utilizada. Com a redução na freqüência de execução destes processos, pode-se obter um ganho significativo no processamento, sem proporcionar uma perda de qualidade no processo adaptativo, principalmente quando o sistema estiver em condições normas de funcionamento.

	IR	IRSN	IRSNT	IRSNMT
CONTROLE	5,50%	5,25%	4,71%	0,96%
MAPA	0,00%	0,00%	0,00%	75,27%
CONTROLE (+MAPA)	5,50%	5,25%	4,71%	76,23%
ED	48,47%	52,07%	49,40%	12,48%
ADAPT	1,40%	2,11%	3,76%	0,98%
DIAG	9,95%	8,61%	10,72%	3,87%
SCHEDULE	8,97%	8,39%	8,93%	2,18%
INIT	0,51%	0,41%	0,51%	0,10%
SIMULADOR	25,22%	23,19%	21,99%	4,16%

Tabela 9.4: Porcentagem do tempo consumido por cada agrupamento de funções.

	Tempo médio (s)	Número de ciclos	Duração média (s)
IR	74,841950	300885	0,00024874
IRSN	79,500127	290672	0,00027350
IRSNT	80,077356	285318	0,00028066
IRSNMT	107,367748	228094	0,00047072

Tabela 9.5: Tempo e ciclo de execução de cada configuração.

Após a avaliação de cada configuração individualmente, foi criada a configuração ADAPT, que ao controle adaptativo ativado realizando trocas de configurações ao longo da execução da missão, como mostrado na Figura 9.7. Nas configurações IR e IRSN nas quais não existem testes de detecção de falhas ativados, a confiabilidade dos sensores é reduzida gradualmente, o que força mesmo na ausência de falhas, um ciclo de adaptações constante (... $\longrightarrow IRSNT \longrightarrow IRSN \longrightarrow IR \longrightarrow IRSN \longrightarrow IRSNT \longrightarrow$ ...).

O índice de desempenho de cada adaptação individual, mostrado na Tabela 9.6, foi calculado em função do tempo médio de execução da missão, sem a inserção de falhas. Os índices de desempenho mostrados na tabela foram utilizados no controle adaptativo, com exceção do associado à configuração ADAPT que foi incluído apenas para facilitar comparações.

Com o intuito de facilitar as comparações, os parâmetros de controle de todas às configurações foram ajustados de forma a para obter um sucesso de 100% na execução da missão sem a presença de falhas. Todas as configurações foram avaliadas inicialmente sem a presença de falhas e depois foram inseridas falhas uma de forma cumulativa até que não se obtenha sucesso algum nas tentativas. Para cada configuração e número de falhas diferentes foram executadas 160 vezes para a obtenção dos dados, totalizando mais de 16000 experimentos.

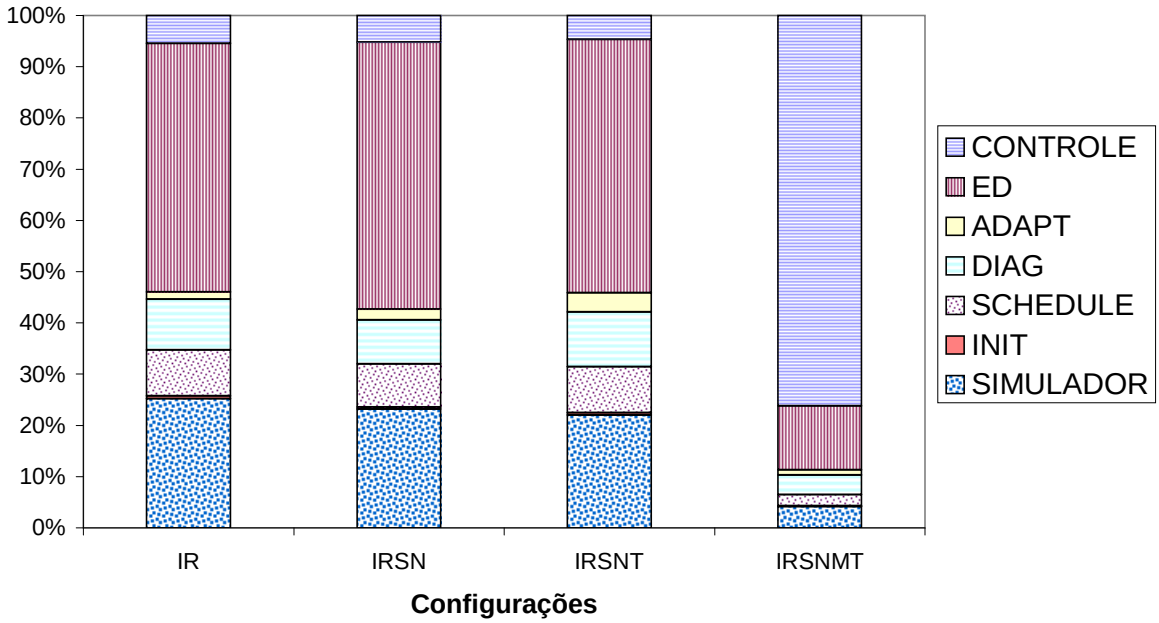


Figura 9.8: Porcentagem de processamento consumido por cada agrupamento.

De todos os experimentos que falharam, apenas 1 foi devido a uma colisão, o que mostra que o controle implementado foi bastante conservador para manter uma distância mínima dos obstáculos. Todos os outros que falharam atingiram o tempo limite de 135 segundos, quando a missão foi abortada. Os tempos utilizados para o cálculo do desempenho das configurações, foram obtidos apenas dos experimentos concluídos com sucesso.

O gráfico mostrado na Figura 9.10 mostra a porcentagem de sucesso na execução da missão em função da configuração utilizada e do número de falhas inseridas. O gráfico mostrado na Figura 9.11 apresenta o tempo médio de execução das missões que completaram com sucesso, para cada configuração variando o número de falhas. O gráfico da Figura 9.12 apresenta a mesma informação, sendo o tempo médio de execução das missões normalizado em função do melhor desempenho obtido. Desta forma fica mais clara a perda de desempenho em função da configuração e do número de defeitos inseridos. Vale ressaltar, que com o aumento do número de defeitos, as missões que completam com sucesso se reduz significativamente, reduzindo também o número de amostras utilizadas para desenhar o gráfico.

A configuração que utiliza o mapa IRSNMT se mostrou inferior em termos de desempenho e muito superior na tolerância a falhas, tendo algum sucesso até o número de 18 defeitos em 32 sensores. O desempenho desta configuração como percebido nos

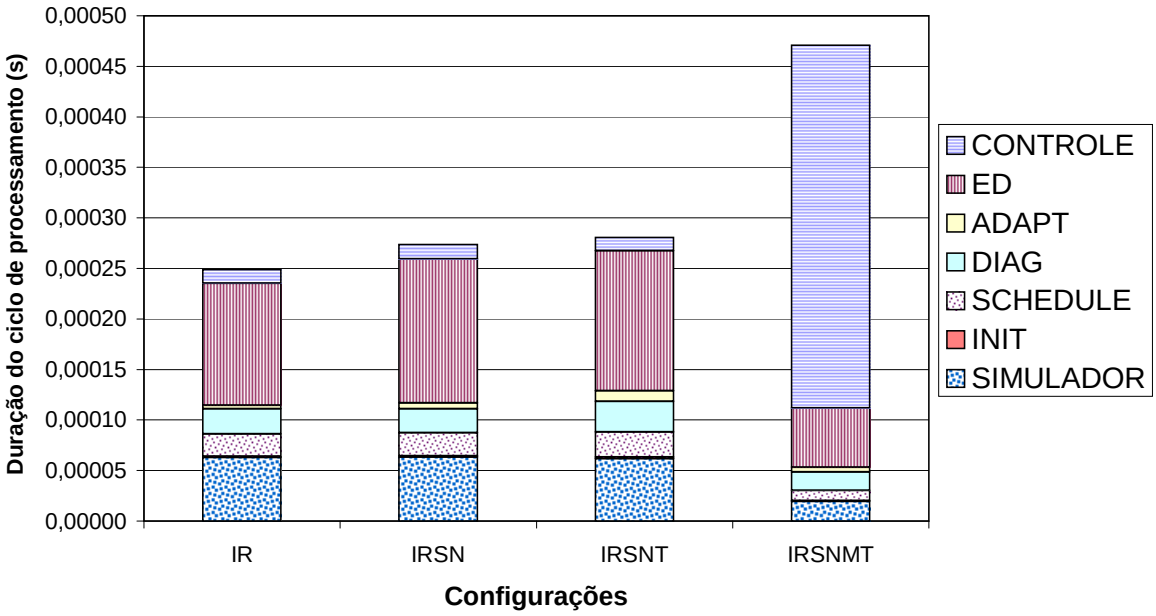


Figura 9.9: Duração do ciclo de processamento de cada configuração, incluindo a porcentagem de cada agrupamento de funções.

gráficos degradou mais suavemente que as outras em função da presença de falhas.

A configuração IR que utiliza somente dos sensores infravermelhos, com apenas 16 sensores ativos, apresentou um melhor desempenho na ausência de falhas, mas se degradou rapidamente. As configurações IRSN e IRSNT mantiveram sempre desempenhos próximos, o que mostra que o uso de informação histórica dos sensores aumenta a chance de sucesso da missão significativamente na presença de falhas.

Os testes com o controle adaptativo ativo (ADAPT) mostraram um resultado que otimiza o desempenho e confiabilidade simultaneamente. A taxa de sucesso em completar as missões da configuração ADAPT, como visto na Figura 9.10, foi superior as configurações IR, IRSN e IRSNT e muito próxima a IRSNMT, como esperado.

O desempenho da configuração ADAPT, mostrado na Figura 9.11 e na Figura 9.12, inicia próximo ao desempenho da configuração IRSNT e se aproxima da curva correspondente a IRSNMT, à medida que o número de defeitos aumenta, significando que o uso desta configuração também aumenta. O comportamento do desempenho corresponde ao esperado, se aproximando ao desempenho da configuração que usa mais redundância à medida que o robô fica menos confiável.

O objetivo da configuração ADAPT foi combinar o desempenho e a confiabilidade. A realização deste objetivo pode ser visto no gráfico da Figura 9.13, no qual

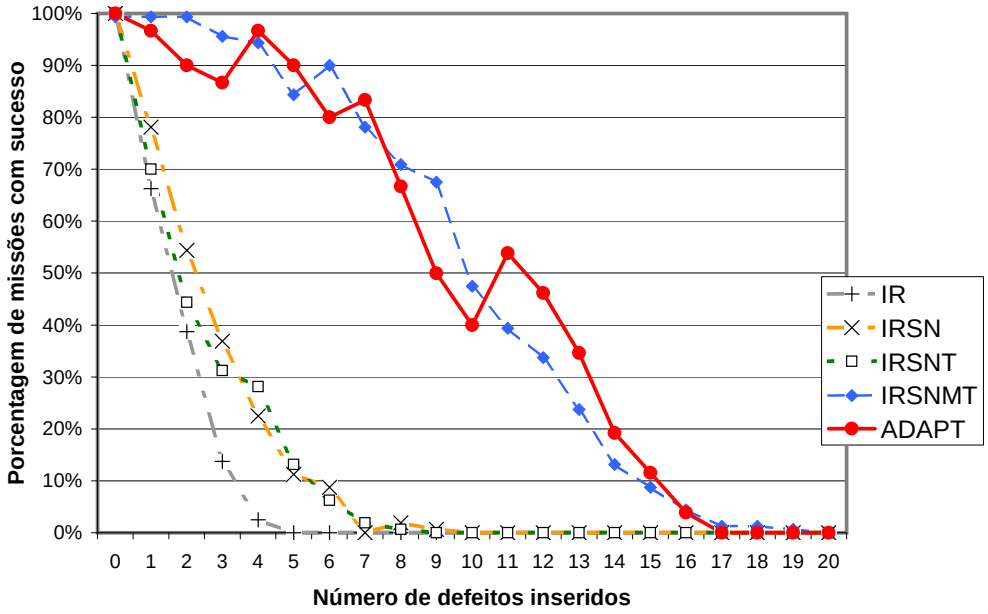


Figura 9.10: Porcentagem de sucesso nas missões pelo número de defeitos inseridos para cada configuração.

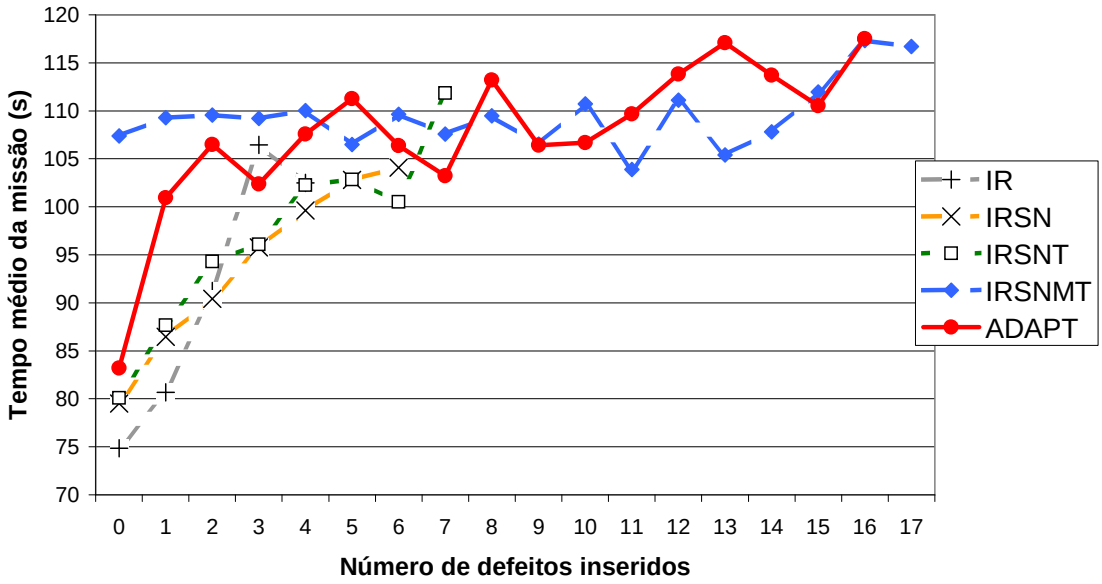


Figura 9.11: Tempo médio de execução das missões com sucesso pelo número de defeitos inseridos.

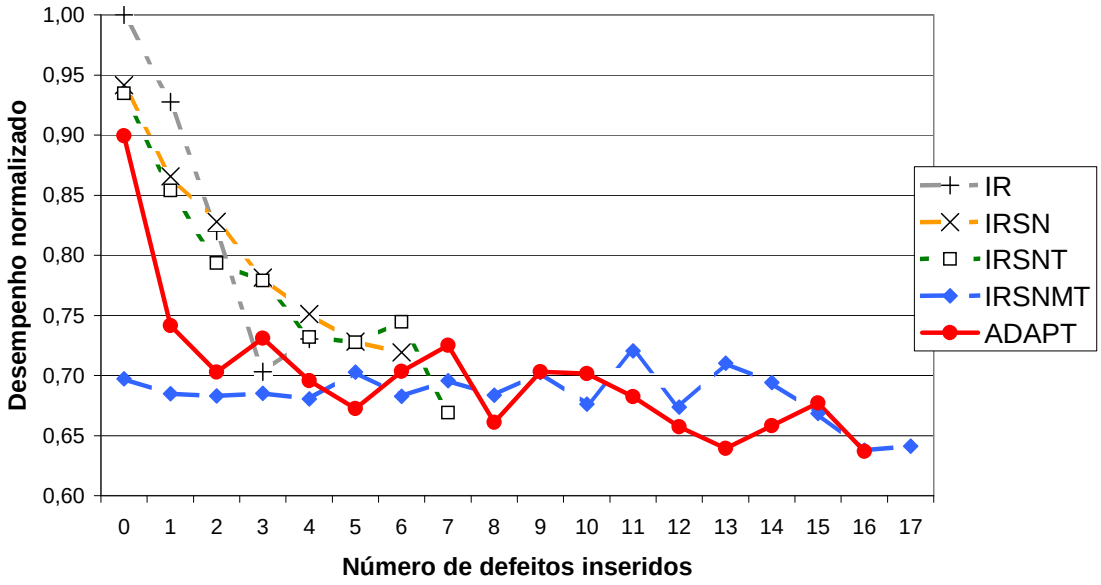


Figura 9.12: Desempenho normalizado em função do tempo de execução mais rápido, variando o número de defeitos inseridos.

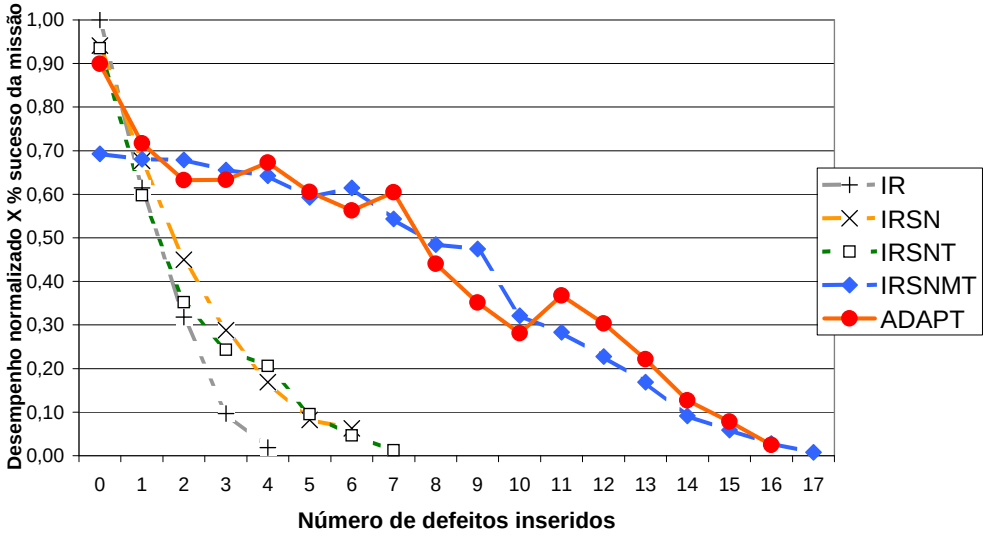


Figura 9.13: Produto do desempenho normalizado e da taxa de sucesso de cada configuração variando o número de defeitos inseridos.

	Tempo médio (s)	**Índice de Desempenho**
IR	74.841950	1.0000
IRSN	79.500127	0.9414
IRSNT	80.077356	0.9346
IRSNMT	107.367748	0.6971
ADAPT	80.291736	0.9321

Tabela 9.6: Tempo médio de execução e índice de desempenho calculado.

a porcentagem de sucesso na execução da missão foi multiplicada pelo desempenho normalizado da mesma. Tanto a porcentagem de sucesso, como o desempenho corresponde a valores na faixa de 0 até 1. Neste caso o produto dos dois fatores também fica restrito a mesma faixa. A configuração ADAPT apresentou um índice melhor para a maioria dos números de defeitos inseridos. O resultado obtido foi satisfatório atendendo os objetivos do protótipo. O ajuste do controle adaptativo se mostrou uma tarefa difícil que requer tempo e muitos experimentos.

Capítulo 10

Conclusões

> Sabedoria é saber qual é a próxima coisa a fazer. Capacidade é saber como fazê-la. Virtude é fazê-la.
>
> David Starr Jordan(1851-1931)

Uma metodologia para facilitar o processo de inserção de tolerância a falhas adaptativa no controle de um robô móvel ou de um outro sistema autômato foi desenvolvida neste trabalho. O mérito maior do trabalho não esta nos detalhes das soluções de implementação ou fórmulas definidas ao longo do texto, e sim no desenvolvimento de uma metodologia completa capaz de facilitar a integração gradual de mecanismos de tolerância a falhas conhecidos em um controle de um sistema complexo como um robô móvel e autônomo.

A tolerância a falhas não oferece a um sistema nenhuma capacidade que este já não possua, com o programa de controle ou configuração adequados. Portanto, um sistema tolerante a falhas é um sistema capaz de selecionar uma configuração apropriada para um dado estado de falhas, ou de efetuar uma ação adequada para se reparar.

O modelo proposto de arquitetura utiliza um controle dividido de forma conceitual em dois níveis. A sua metodologia de desenvolvimento facilita a criação e gerenciamento de múltiplos fluxos de controle de baixo nível funcionalmente equivalentes. A estrutura padronizada de conexão no fluxo favorece o tratamento de informações redundantes e o uso de bibliotecas de funções de detecção de falhas com dados de natureza distinta.

A estrutura de fluxo do baixo nível permitiu propor um método simples para previsão da confiabilidade de cada configuração gerada. Além disso, esta estrutura

facilita a utilização de métodos de diagnóstico, como as redes Baysianas, que empregam diretamente relações de causalidade e dependências.

O alto nível de controle é integrado totalmente com os mecanismos de tolerância a falhas através da síntese do Grafo de Controle Adaptativo. Este grafo concentra as diversas informações necessárias para realizar as transições de fase, o processo adaptativo e as recuperações de falhas de forma direta ou indireta. Com o uso deste grafo o problema de composição de diversas funcionalidades complexas fica reduzido ao uso de testes bem simples vinculados a arestas e métodos de busca clássicos em grafos. Além disso, um algoritmo possível de ser automatizado é descrito ao longo dos capítulos 5 e 8. A implementação do protótipo apresentou uma reduzida utilização de memória demonstrando a viabilidade da utilização do grafo inclusive para projetos maiores.

Um conjunto de modos de operação da Plataforma de Controle Adaptativo é proposto de forma a facilitar o desenvolvimento de um projeto de maior porte, incluindo a coleta de dados que facilitem os ajustes mais complexos do sistema. Entre eles o controle adaptativo e a detecção de falhas.

A metodologia proposta é compatível com mecanismos de tolerância a falhas já existentes e utilizados em outros sistemas. Além disso, a estruturação final do programa de controle é plenamente adequada para os usos em sistemas multiprocessados.

A metodologia proposta considerou sempre a possibilidade de utilizar um ambiente próprio de desenvolvimento com um *Framework*. Muitos pontos do modelo, como a criação do Grafo de Controle Adaptativo, foram definidos com o intuito de permitir a automatização, sendo complexos para uma implementação totalmente manual. As várias estruturas de dados que existem no modelo são interrelacionadas, o que torna complexo o processo de alteração e redefinição pelo projetista. O uso de mecanismos automáticos de verificação de regras de coerência entre estas estruturas e entre os modelos codificados pode simplificar o projeto para o projetista, além de evitar erros simples de desatenção ou esquecimento.

O processo de desenvolvimento proposto prevê uma relação estreita entre as várias etapas distintas, incluindo a seleção de parâmetros e a realização de ajustes dos mesmos. Estas relações que existem em várias metodologias, podem tornar o desenvolvimento extremamente interativo e até repetitivo. Esta característica evidencia a necessidade controles adequados de versões tanto dos módulos codificados, quanto dos arquivos de definição de parâmetros e outras configurações existentes.

O protótipo validou de forma satisfatória o modelo desenvolvido, mas levantou algumas questões importantes. A implementação do acesso aos *EDs* que é um ponto chave no processo de síntese das diversas adaptações do sistema tem um custo de processamento significativo. Este custo pode inviabilizar o uso da estrutura de reconfiguração para controles simples, como ocorre nos controles de abordagem reativa.

Outro ponto importante reforçado pela análise dos dados obtidos do protótipo é

que o ganho que conduz o controle adaptativo deve ser ajustado a cada aplicação específica. O cálculo deste fator é fundamental para o sucesso da abordagem adaptativa, mesmo que o sistema possua um conjunto de adaptações abrangente. A metodologia necessita de ferramentas de análise adequadas para comparar e correlacionar as variações de confiabilidade e desempenho permitindo a definição dos critérios adaptativos apropriados.

Resumindo estas diversas considerações, a metodologia desenvolvida atendeu os objetivos propostos. Entretanto, para o desenvolvimento de projetos de grande porte se faz necessário ferramentas que auxiliem o projetista.

10.1 Trabalhos futuros

Ao longo do texto fica clara a existência de os vários pontos que podem ser ainda desenvolvidos ou aprofundados a partir deste trabalho inicial da tese. Muitos deles são comuns a qualquer sistema que se proponha a implementar a tolerância a falhas adaptativa.

Um dos pontos importantes específicos para o modelo desenvolvido é a extensão do mesmo para tratar a ativação e desativação de tarefas de tempo real. O modelo deverá futuramente incluir o uso de políticas de replicação e redundância de tarefas de tempo real. A melhor abordagem para esta questão é buscar a integração com algum sistema que já ofereça as funcionalidades básicas de tolerância a falhas para tarefas de tempo real e comunicação confiável entre processadores, como por exemplo, o trabalho realizado [OLIVEIRA et al., 2003] neste mesmo departamento. Além disso, é interessante também estender o processo de síntese das políticas de redundância para incluir opções específicas de detecção e tratamento de falhas de software no processamento no fluxo.

Outra extensão desejável para o trabalho realizado é a possibilidade de executar paralelamente fluxos de processamento com durações distintas como foi descrito na Seção 6.2. Este tipo de funcionalidade pode simplificar a programação do projetista de tarefas que necessitem serem realizadas com restrições temporais de ordens de grandezas diferentes. Como foi descrito no texto, embora a simplicidade dos *BFs* e *EDs* seja mantida, a implementação da comunicação e escalonamento da *PCA* e as funções da *API* utilizadas para programação ficam muito mais complexas.

No modelo desenvolvido, os *EDs* podem ser considerados pontos de teste ou depuração sendo extremamente fácil exportar os valores processados no fluxo para arquivos de registro. Estes dados podem ser analisados externamente ao controle procurando pela existência de correlações entre eles. As informações apuradas desta forma podem ser utilizadas para refinamento do próprio controle desenvolvido, ou das tarefas de detecção de falhas e de métodos de diagnósticos.

A eficácia do modelo desenvolvido se baseia na precisão de seus índices internos para permitir os processo de otimização e detecção de falhas indiretas. O cálculo dos índices de sucesso ou de desempenho de qualquer ação realizada por um robô é uma tarefa extremamente complexa, sendo o mesmo tema de muitos trabalhos na literatura. Além disso, abordagens de otimizações estáticas muitas vezes se mostram inadequadas ou ineficazes quando são consideradas missões de maior duração, nas quais os desgastes ou ajustes do sistema interferem na interação com o meio. Um ponto importante para o trabalho seria incorporar métodos de cálculo dinâmico ou de aprendizado destes parâmetros diretamente no controle implementado. De forma que, com a utilização do sistema seria possível refinar suas capacidades adaptativas e otimizar o seu desempenho progressivamente.

Os objetivos de uma fase podem ser traduzidos para o índice de confiança de uma determinada configuração (I^C) através da associação de pesos a confiança das informações atribuídas a determinados *EDs*. O cálculo realizado levou em conta apenas a probabilidade de falhas nos dados coletados, não considerando a influência direta da precisão destes na probabilidade de sucesso da fase. Este é um dos pontos mais importantes a serem desenvolvidos.

A fórmula utilizada para a otimização do ganho não incluiu o custo de efetuar uma reconfiguração no sistema. Este é fator muito importante e deve ser tratado futuramente, pois é essencial quando se necessita ativar ou desativar elementos de hardware específicos.

A síntese de vários destes possíveis trabalhos futuros, como já foi dito é o desenvolvimento de ferramentas de auxílio ao projeto utilizando a metodologia. Alguns dos principais requisitos deste conjunto de ferramentas foram destacados a seguir:

- Possibilite a verificação de coerência das estruturas de dados e códigos utilizados ou permitam a síntese automática a partir de representações gráficas de mais alto nível.
- Permita a coleta de dados internos externos do funcionamento do controle e realize análise e correlações entre os dados identificando parâmetros de controle ou de detecção de falhas adequados.
- Correlacione os resultados da execução de várias adaptações facilitando a composição das funções que calculam o sucesso esperado e desempenho, facilitando o ajuste da função de ganho esperado.
- Mantenha o controle completo sobre as versões de desenvolvimento e versões de ajustes.
- Implemente todas as funcionalidades descritas no ciclo de projeto definido na Seção 6.8.

- Analise aspectos estruturais do controle e síntese automaticamente uma versão do código para executar fórmulas e os cálculos dos índices existentes no modelo.

Como ficou claro ao longo deste texto de tese, o trabalho desenvolvido é apenas o começo de um longo caminho para se desenvolver de forma simples e eficaz o controle de sistemas autônomos com características de tolerância a falhas adaptativa.

Referências Bibliográficas

[Arbib, 1992] Arbib, M. (1992). Schema theory. *Encyclopedia of Artificial Intelligence*, 2:1427–1443. 2nd Edition.

[Arkin, 1989] Arkin, R. C. (1989). Towards the unification of navigational planning and reactive control. In *AAAI Spring Symposium on Robot Navigation*, pages 1–5.

[Arkin, 1995] Arkin, R. C. (1995). Intelligent robotic systems. *IEEE Expert*, 10(2):6–8.

[Arkin, 1998] Arkin, R. C. (1998). *Behavior-Based Robotics*. Mit Press.

[Bagchi, 2001] Bagchi, S. (2001). *Hierarchical error detection in a software implemented fault tolerance (SIFT) environment*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois.

[Bagchi et al., 1998] Bagchi, S., Whisnant, K., Kalbarczyk, Z., and Iyer, R. (1998). The chameleon infrastructure for adaptive, software implemented fault tolerance. In *Symposium on Reliable Distributed Systems*.

[Barabanov, 1997] Barabanov, M. (1997). A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining abd Technology, Socorro, New Mexico.

[Barreto and Fernandes, 1997] Barreto, R. S. and Fernandes, A. O. (1997). Um estudo de técnicas para tolerância a falhas em sistemas distribuídos. Technical report, Departamento de Ciência da computação. Relatório Técnico DCC-RT 001/97.

[Bertossi et al., 1999] Bertossi, A. A., Mancini, L. V., and Rossini, F. (1999). Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):934–945.

[Birk and Kenn, 2001] Birk, A. and Kenn, H. (2001). Efficient scheduling of behavior-processes on different time-scales. In *2001 IEEE International Conference on Robotics and Automation*, Seoul, Korea.

[Brooks, 1985] Brooks, R. A. (1985). A robust layered control system for a mobile robot. Technical report, MIT AI Lab. Memo 864.

[Brooks, 1986] Brooks, R. A. (1986). Achieving artificial intelligence through building robots. Technical Report 899, MIT AI Lab.

[Brooks, 1989a] Brooks, R. A. (1989a). How to build complete creatures rather than isolated cognitive simulators. *Architectures for Intelligence*, pages 225–239.

[Brooks, 1989b] Brooks, R. A. (1989b). A robot that walks; emergent behaviors from a carefully evolved network. Technical report, MIT AI Lab.

[Brooks, 1990] Brooks, R. A. (1990). The behavior language user's guide. Technical report, MIT AI Lab. Memo 1227.

[Brooks, 1991a] Brooks, R. A. (1991a). Integrated systems based on behaviors. *SIGART Bulletin*, 2(4):46–50.

[Brooks, 1991b] Brooks, R. A. (1991b). Intelligence without reason. Technical report, MIT AI Lab Memo 1293.

[Brooks, 1991c] Brooks, R. A. (1991c). Intelligence without representation. *Artificial Intelligence Journal*, 47:pp. 139–159.

[Brooks, 1991d] Brooks, R. A. (1991d). New approaches to robotics. *Science*, 253:1227–1232.

[Brooks, 1999] Brooks, R. A. (1999). *Cambrian Intelligence*. The MIT Press Books.

[Bryson, 2000] Bryson, J. J. (2000). The study of sequential and hierarchical organisation of behaviour via artificial mechanisms of action selection. Master's thesis, University of Edinburgh.

[Cavallaro and Walker, 1994] Cavallaro, J. R. and Walker, I. D. (1994). A survey of nasa and military standards on fault tolerance and reliability applied to robotics. In *AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space*, pages 282–286, Houston, TX.

[Chaimowicz et al., 2001] Chaimowicz, L., Sugar, T., Kumar, V., and Campos, M. F. M. (2001). An architecture for tightly coupled multi-robot cooperation. In *IEEE International Conference on Robotics and Automation*.

[Dias and Stentz, 2000] Dias, M. B. and Stentz, A. T. (2000). A free market architecture for distributed control of a multirobot system. In *6th International Conference on Intelligent Autonomous Systems (IAS-6)*, pages 115–122.

[Ferrell, 1993] Ferrell, C. (1993). Robust agent control of an autonomous robot with many sensors and actuators. Master's thesis, Massachussets Institute of Technolgy.

[Ferrell, 1994] Ferrell, C. (1994). Failure recognition and fault tolerance of an autonomous robot. Technical report, Artificial Intelligence Laboratory.

[Firby, 1994] Firby, R. J. (1994). Task networks for controlling continuous processes. In *Second International Conference on AI Planning Systems (AAAI Press, 1994)*, Chicago, IL.

[Firby et al., 1995] Firby, R. J., Prokopowicz, P., and Swain, M. (1995). Plan representations for picking up trash. In *International Joint Conference on Artiøcial Intelligence, IJCAI'95*.

[Fohler, 1997] Fohler, G. (1997). Adaptive fault-tolerance with statically scheduled real-time systems. In *9th Euromicro Workshop on Real Time Systems–1997*.

[Gerkey and Mataric, 2000] Gerkey, B. P. and Mataric, M. J. (2000). Principled communication for dynamic multi-robot task allocation. In *International Symposium on Experimental Robotics 2000*, pages 341–352., Waikiki, Hawaii.

[Gerkey and Mataric, 2001] Gerkey, B. P. and Mataric, M. J. (2001). Pusher-watcher: An approach to fault-tolerant tightly-coupled robot coordination. Technical report, nstitute for Robotics and Intelligent Systems. Technical Report IRIS-01-403.

[Goel et al., 2000] Goel, P., Dedeoglu, G., Roumeliotis, S. I., and Sukhatme, G. S. (2000). Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *IEEE International Conference on Robotics and Automation*, pages 2302–2309, San Francisco, California.

[Goldberg and Mataric, 2000] Goldberg, D. and Mataric, M. J. (2000). Robust behavior-based control for distributed multi-robot collection tasks. Technical report, USC Institute for Robotics and Intelligent Systems. Technical Report IRIS-00-387.

[Gonzalez et al., 1997] Gonzalez, O., Shrikumar, H., Stankovic, J. A., and Ramamritham, K. (1997). Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *18th IEEE Real-Time Systems Symposium (RTSS '97)*.

[Hamilton et al., 2001] Hamilton, K., Lane, D. M., K.Taylor, N., and Brown, K. (2001). Fault diagnosis on autonomous robotic vehicles with recovery: An integrated heterogeneous-knowledge approach. In *2001 IEEE International Conference on Robotics and Automation*, Seoul, Korea.

[Hecht et al., 2000] Hecht, M., Hecht, H., and Shokri, E. (2000). Adaptive fault tolerance for spacecraft. In *IEEE Aerospace 2000 Conference*, Big Sky, MT.

[Horswill, 1994] Horswill, I. (1994). *Specialization of Perceptual Processes*. PhD thesis, Massachusetts Institute of Technology.

[Huntsberger, 1998] Huntsberger, T. L. (1998). Fault tolerant action selection for planetary rover control. In *SPIE Symposium on Sensor Fusion and Decentralized Control in Robotic Systems*, volume 3523, pages 150–156, Boston, MA.

[Huntsberger et al., 2000] Huntsberger, T. L., Aghazarian, H., Baumgartner, E., and Schenker, P. S. (2000). Behavior-based control systems for planetary autonomous robot outposts. In *IEEE AEROSPACE 2000*, pages 679–685, Big Sky, Montana.

[Jagt, 2002] Jagt, R. M. (September 2002). Support for multiple cause diagnosis with bayesian networks. Master's thesis, Delft University of Technology.

[Kalbarczyk et al., 1999] Kalbarczyk, Z. T., Iyer, R. K., Bagchi, S., and Whisnant, K. (1999). Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579.

[Kalman, 1960] Kalman, Rudolph, E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45.

[Kandasamy and Hayes, 1998] Kandasamy, N. and Hayes, J. P. (1998). Tolerating transient faults in statically scheduled safety-critical embedded systems. In *18th IEEE Symposium on Reliable Distributed Systems*.

[Kim, 2000] Kim, K. (2000). Object-oriented real-time distributed programming and support middleware. In *ICPADS 2000 (7th Int'l Conf. on Parallel & Distributed Systems)*, Iwate, Japan. IEEE CS Press.

[Kim and Lawrence, 1992] Kim, K. and Lawrence, T. (1992). Adaptive fault-tolerance in complex real-time distributed computer system applications. *Computer Communications*, 15(4):243–251.

[Koji et al., 2001] Koji, I., Nokata, M., and Ishii, H. (2001). General danger-evaluation method of human-care robot control and development of special simulator. In *2001 IEEE International Conference on Robotics and Automation*, Seoul, Korea.

[Laplante, 1997] Laplante, P. A. (1997). *Real-Time Systems Design and Analisis*. IEEE Press.

[Lewis and Maciejewski, 1994] Lewis, C. L. and Maciejewski, A. A. (1994). Dexterity optimization of kinematically redundant manipulators in the presence of joint failures. *Computers and Electrical Engineering*, 20(3):273–288.

[Liberato et al., 2000] Liberato, F., Melhem, R., and Mossé, D. (2000). Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transactions on Computers*, 49(9):906–914.

[Liu, 2001] Liu, G. (2001). Control of robot manipulators with consideration of actuator performance degradation and failures. In *2001 IEEE International Conference on Robotics and Automation*, Seoul, Korea.

[Lueth and Laengle, 1994] Lueth, T. C. and Laengle, T. (1994). Fault-tolerance and error recovery in an autonomous robot with distributed controlled components. In Asama, H., Fukuda, T., Arai, T., and Endo, I., editors, *Distributed Autonomous Robotic Systems*, Springer-Verlag.

[Maes, 1989a] Maes, P. (1989a). The dynamics of action selection. In *International Joint Conference on Artificial Intelligence*, volume 2, pages 991–998, Detroit.

[Maes, 1989b] Maes, P. (1989b). How to do the right thing. *Connection Science Journal*, 1(3):291–323. Also MIT AI-Memo 1180.

[Maes, 1990] Maes, P. (1990). Situated agents can have goals. In *Designing Autonomous Agents*, pages 49–70.

[Maes and Brooks, 1990] Maes, P. and Brooks, R. A. (1990). Learning to coordinate behaviors. In *AAAI, Boston, MA*, pages 796–802.

[Marzwell et al., 1994] Marzwell, N. I., Tso, K. S., and Hecht, M. (1994). An integrated fault tolerant robotic control system for high reliability and safety. In *Technology 2004*, Washington, DC.

[Mataric, 1992a] Mataric, M. J. (1992a). Behavior-based systems: Main properties and implications. In *IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, pages 46–54, Nice, France.

[Mataric, 1992b] Mataric, M. J. (1992b). Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312.

[Mataric, 1994] Mataric, M. J. (1994). *Interaction and Intelligent Behavior*. PhD thesis, Massachusetts Institute of Technology.

[Mataric, 1997] Mataric, M. J. (1997). Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence, special issue on Software Architectures for Physical Agents*, 9(2-3):323–336.

[Mili et al., 1998] Mili, A., Cukic, B., Xia, T., and Ayed, R. B. (1998). Combining fault avoidance, fault removal and fault tolerance: An integrated model. In *14th IEEE International Conference on Automated Software Engineering*.

[Murphy, 1994] Murphy, R. R. (1994). *Sensor Fusion*, pages 857–860. MIT Press, Cambridge, MA. in Arbib, M.A, The Handbook of Brain Theory and Neural Networks.

[Murphy and Hershberger, 1996] Murphy, R. R. and Hershberger, D. (1996). Classifying and recovering from sensing failures in autonomous mobile robots. In *AAAI-96*, pages 922–929., Portland, OR.

[Murphy and Hershberger, 1999] Murphy, R. R. and Hershberger, D. (1999). Handling sensing failures in autonomous mobile robots. *International Journal of Robotics Research*, 18(4):382–400.

[NETO et al., 2003] NETO, D. O. G., JUNIOR, W. M., NOGUEIRA, D. L., and BRAGA, R. P. (2003). Mpm: Middleware para gerência de energia em clusters web. In *Simpósio Brasileiro de Redes de Computadores*, pages 281–295, Natal, Rio Grande do Norte.

[Nicolescu and Mataric, 2000a] Nicolescu, M. and Mataric, M. J. (2000a). Extending behavior-based systems capabilities using an abstract behavior representation. Technical report, USC Institute for Robotics and Intelligent Systems. IRIS Technical Report IRIS-00-389.

[Nicolescu and Mataric, 2000b] Nicolescu, M. N. and Mataric, M. J. (2000b). Deriving and using abstract representation in behavior-based systems. In *American Association for Artificial Intelligence*.

[Nikovski, 2000] Nikovski, D. (2000). Constructing bayesian networks for medical diagnosis from incomplete and partially correct statistics. *IEEE Transactions on Knowledge and Data Engineering*, 12(14):509 – 516. special issue on constructing Bayesian networks.

[Nilsson, 1969] Nilsson, N. (1969). A mobile automation: An application of artificial intelligence techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 509–520, Washington, D.C.

[Nomadic, 1997a] Nomadic, T. (1997a). *Nomad 200 Hardware Manual*. Nomadic Techologies Inc.

[Nomadic, 1997b] Nomadic, T. (1997b). *Nomad 200 Language Reference*. Nomadic Techologies Inc.

[Nomadic, 1997c] Nomadic, T. (1997c). *Nomad 200 User Manual*. Nomadic Techologies Inc.

[OLIVEIRA et al., 2003] OLIVEIRA, M. P. D., Fernandes, A. O., CAMPOS, S., and ZUQUIM, A. L. D. A. P. (2003). Garanteeing fault tolerance though scheduling on a can bus. In *Workshop de Testes e Tolerância a Falhas*, pages 43–50, Natal, Rio Grande do Norte.

[Paredis and Khosla, 1994] Paredis, C. J. and Khosla, P. K. (1994). Mapping tasks into fault tolerant manipulators. In *IEEE International Conference on Robotics and Automation*, San Diego, CA.

[Paredis and Khosla, 1996] Paredis, C. J. and Khosla, P. K. (1996). Fault tolerant task execution through global trajectory planning. In *Reliability Engineering and System Safety*.

[Parker, 1994] Parker, L. E. (1994). *Heterogeneous Multi-Robot Cooperation*. PhD thesis, Massachusetts Institute of Technology. AITR-1465.

[Parker, 1997] Parker, L. E. (1997). L-alliance: Task-oriented multi-robot learning in behavior-based systems. *Advanced Robotics, Special Issue on Selected Papers from IROS '96*, 11(2):305–322.

[Parker, 1999] Parker, L. E. (1999). Adaptive heterogeneous multi-robot teams. In *Neurocomputing, special issue of NEURAP '98: Neural Networks and Their Applications*, volume 28,, pages 75–92.

[Parker, 2000b] Parker, L. E. (2000b). Current state of the art in distributed autonomous mobile robotics. In L. E. Parker, G. Bekey, J. B., editor, *Distributed Autonomous Robotic Systems 4*, pages 3–12, Springer-Verlag Tokyo 2000.

[Parker, 2000a] Parker, L. E. (March 2000a). The alliance architecture for multi-robot control. NASA Surface Systems Quarterly Meeting.

[Payton et al., 1992] Payton, D., Keirsey, D., Kimple, D., Krozel, J., and Rosenblatt, K. (1992). Do whatever works: A robust approach to fault-tolerant autonomous control. *Applied Intelligence*, 2:225–250.

[Pearl, 1988] Pearl, J. (1988). *Probalilistc Reasoning In Intelligent Systems: Networks of Plausible Inferente*. Ronald J. Brackman (AT&T Bell Laboratories), San Mateo, California.

[Pirjanian, 1999] Pirjanian, P. (1999). Behavior coordination mechanisms - state-of-the-art. Technical report, USC Robotics Research Laboratory, Los Angels,CA.

[Przytula and Thompson, 2000] Przytula, K. and Thompson, D. (2000). Construction of bayesian networks for diagnosis. In Press, I. C. S., editor, *Proceedings of the 2000 IEEE Aerospace Conference*, volume 5, pages 193–200.

[Randell and Xu, 1995] Randell, B. and Xu, J. (1995). *The Evolution of the Recovery Block Concept, Software Fault Tolerance*. Willey. Lye M.R. (edt).

[Redimbo, 1998] Redimbo, G. R. (1998). Generalized algorithm-based fault tolerance: Error correction via kalman estimation. *IEEE Transactions on Computers*, 47(6):639–655.

[Rosenblatt, 1997] Rosenblatt, J. (1997). *DAMN: A Distributed Architecture for Mobile Navigation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA. Technical Report CMU-RI-TR-97-01.

[Roumeliotis et al., 1998a] Roumeliotis, S. I., Sukhatme, G. S., and Bekey, G. A. (1998a). Fault detection and identification in a mobile robot using multiple-model estimation. In *IEEE International Conference on Robotics and Automation*, pages 2223–2228, Leuven, Belgium.

[Roumeliotis et al., 1998b] Roumeliotis, S. I., Sukhatme, G. S., and Bekey, G. A. (1998b). Sensor fault detection and identification in a mobile robot. In *1998 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1383–1388., Victoria, Canada,.

[Schneider-Fontan and Mataric, 1998] Schneider-Fontan, M. and Mataric, M. J. (1998). Territorial multi-robot task division. *IEEE Transactions on Robotics and Automation*, 14(5):815–822.

[Shokri and Beltas, 2000] Shokri, E. and Beltas, P. (2000). An experiment with adaptive fault tolerance in highly-constraint systems. In *Fifth International Workshop on Object-Oriented Real-Time Dependable Systems*.

[Shokri et al., 1998] Shokri, E., Crane, P., Kim, K. H., and Subbaraman, C. (1998). Architecture of roafts/solaris: A solaris-based middleware for real-time object-oriented adaptive fault tolerance support. In *COMPSAC '98 (IEEE CS 22nd Int'l Computer Software and Applications Conf.)*, pages 90–98., Vienna, Austria.

[Somani and Vaidya, 1997] Somani, A. K. and Vaidya, N. H. (1997). Understanding fault tolerance and reliability. *IEEE Transactions on Computers*, 30(4):45–50.

[Sun and McCartney, 2001] Sun, H. and McCartney, R. (2001). A binary tree based approach for the design of fault-tolerant robot team. In *FLAIRS-2001*, Key West, Florida.

[Vemuri and Polycarpou, 1997] Vemuri, A. T. and Polycarpou, M. M. (1997). Robust nonlinear fault diagnosis in input-output systems. *International Journal of Control*, 68:343–360,.

[Visinsky, 1994] Visinsky, M. (1994). *Dynamic Fault Detection and Intelligent Fault Tolerance Methods for Robotics*. PhD thesis, Rice University.

[Walter, 1950] Walter, W. G. (1950). An imitation of life. *Scientific American*, 182(5):42–45.

[Weber et al., 2000] Weber, S., Jenkins, C., and Mataric, M. J. (2000). Imitation using perceptual and motor primitives. In *Autonomous Agents 2000*, pages 136–137, Barcelona, Spain.

[Werger, 1999] Werger, B. B. (1999). Cooperation without deliberation: A minimal behavior-based approach to robot team. *Artificial Intelligence*, 110:293–320.

[Werger, 2000] Werger, B. B. (2000). Ayllu: Distributed port-arbitrated behavior-based control. In Parker, L. E., Bekey, G., and Barhen, J., editors, *Distributed Autonomous Robot Systems 2000*, pages 25–34, Springer Verlag, (Knoxville, TN).

Apêndice A

Elementos de Dados do protótipo

Este apêndice apresenta a estrutura de dados estática contendo algumas informações sobre os Elementos de Dados (*EDs*) e sobres os Parâmetros de Controle (*PCs*).

```
/* Eds Associados ao hardware */
{"R_STATE_IR_0",R_STATE_IR_0,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_IR_1",R_STATE_IR_1,L_D,(ValueDataUnion)(0L),0},
 ...
{"R_STATE_IR_15",R_STATE_IR_15,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_SONAR_0",R_STATE_SONAR_0,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_SONAR_1",R_STATE_SONAR_1,L_D,(ValueDataUnion)(0L),0},
...
{"R_STATE_SONAR_15",R_STATE_SONAR_15,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_BUMPER",R_STATE_BUMPER,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_CONF_X",R_STATE_CONF_X,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_CONF_Y",R_STATE_CONF_Y,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_CONF_STEER",R_STATE_CONF_STEER,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_CONF_TURRET",R_STATE_CONF_TURRET,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_VEL_TRANS",R_STATE_VEL_TRANS,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_VEL_STEER",R_STATE_VEL_STEER,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_VEL_TURRET",R_STATE_VEL_TURRET,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_MOTOR_STATUS",R_STATE_MOTOR_STATUS,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_LASER",R_STATE_LASER,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_COMPASS",R_STATE_COMPASS,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_ERROR",R_STATE_ERROR,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_VOLT_CPU_BAT",R_STATE_VOLT_CPU_BAT,L_D,(ValueDataUnion)(0L),0},
{"R_STATE_VOLT_MOTOR_BAT",R_STATE_VOLT_MOTOR_BAT,L_D,
                (ValueDataUnion)(0L),0},
{"R_STATE_TIME",R_STATE_TIME,L_D,(ValueDataUnion)(0L),0},
```

```
{"R_STATE_POS_DATA_SONAR",R_STATE_POS_DATA_SONAR,P_D,
                (ValueDataUnion)(long)(16*sizeof(PosData)),0},
{"R_STATE_POS_DATA_IR",R_STATE_POS_DATA_IR,P_D,
                (ValueDataUnion)(long)(16*sizeof(PosData)),0},

/* system EDS */
{"SYS_CYCLES_IN_NODE",SYS_CYCLES_IN_NODE,UL_D,(ValueDataUnion)(0L),0},
{"SYS_TIME_IN_NODE",SYS_TIME_IN_NODE,D_D,(ValueDataUnion)(0.0),0},
{"SYS_GLOBAL_CONF",SYS_GLOBAL_CONF,D_D,(ValueDataUnion)(0.0),0},
{"SYS_MISSION_TIME",SYS_MISSION_TIME,D_D,(ValueDataUnion)(0.0),0},

/* ed virtuais */ {"V_ALIGN",V_ALIGN,L_D,(ValueDataUnion)(0L),1},
{"V_ARRIVE",V_ARRIVE,L_D,(ValueDataUnion)(0L),1},
{"V_COLISION",V_COLISION,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_0",V_DIST_0 ,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_1",V_DIST_1 ,L_D,(ValueDataUnion)(0L),1},
...
{"V_DIST_15",V_DIST_15 ,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_FRONT",V_DIST_FRONT,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_LEFT",V_DIST_LEFT,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_NEAR",V_DIST_NEAR,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_NEAR_IDX",V_DIST_NEAR_IDX,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_RIGHT",V_DIST_RIGHT ,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_GOAL",V_DIST_GOAL,L_D,(ValueDataUnion)(0L),1},
{"V_DIST_PI",V_DIST_PI,L_D,(ValueDataUnion)(0L),1},
{"V_LOWBAT",V_LOWBAT,L_D,(ValueDataUnion)(0L),0},
{"V_OBSTACLE",V_OBSTACLE,L_D,(ValueDataUnion)(0L),1},
{"V_CONNECTED",V_CONNECTED,L_D,(ValueDataUnion)(0L),0},
{"R_DIST_0",R_DIST_0 ,L_D,(ValueDataUnion)(0L),1},
{"R_DIST_1",R_DIST_1 ,L_D,(ValueDataUnion)(0L),1},
...
{"R_DIST_15",R_DIST_15 ,L_D,(ValueDataUnion)(0L),1},
{"V_GOAL_ANG",V_GOAL_ANG ,L_D,(ValueDataUnion)(0L),0},
{"V_CHANGE_TEST",V_CHANGE_TEST ,L_D,(ValueDataUnion)(0L),0},

/* Atuadores */
{"A_STATE_VEL_TURRET",A_STATE_VEL_TURRET,L_D,(ValueDataUnion)(0L),0},
{"A_STATE_VEL_STEER",A_STATE_VEL_STEER,L_D,(ValueDataUnion)(0L),0},
{"A_STATE_VEL_TRANSLATION",A_STATE_VEL_TRANSLATION,L_D,
                (ValueDataUnion)(0L),0},
{"A_STATE_DTK",A_STATE_DTK,STR_D,(ValueDataUnion)("Teste"),0},
```

```
/* Parametros de controle */
{"P_BEST_DIST",P_BEST_DIST,L_D,(ValueDataUnion)(0L),0},
{"P_BEST_X",P_BEST_X,L_D,(ValueDataUnion)(0L),0},
{"P_BEST_Y",P_BEST_Y,L_D,(ValueDataUnion)(0L),0},
{"P_DESIRE_DISTANCE",P_DESIRE_DISTANCE,L_D,(ValueDataUnion)(0L),0},
{"P_DIST_PI",P_DIST_PI,L_D,(ValueDataUnion)(0L),0},
{"P_GOAL_X",P_GOAL_X,L_D,(ValueDataUnion)(0L),0},
{"P_GOAL_Y",P_GOAL_Y,L_D,(ValueDataUnion)(0L),0},
{"P_I_X",P_I_X,L_D,(ValueDataUnion)(0L),0},
{"P_I_Y",P_I_Y,L_D,(ValueDataUnion)(0L),0},
{"P_MAX_ANG_ERR",P_MAX_ANG_ERR,L_D,(ValueDataUnion)(0L),0},
{"P_MAX_DIST_ERR",P_MAX_DIST_ERR,L_D,(ValueDataUnion)(0L),0},
{"P_MAX_STEER",P_MAX_STEER,L_D,(ValueDataUnion)(0L),0},
{"P_MAX_TRANS",P_MAX_TRANS,L_D,(ValueDataUnion)(0L),0},
{"P_MIN_DIST",P_MIN_DIST,L_D,(ValueDataUnion)(0L),0},
{"P_MIN_FRONT",P_MIN_FRONT,L_D,(ValueDataUnion)(0L),0},
{"P_LIMIT_TIME",P_LIMIT_TIME,L_D,(ValueDataUnion)(0L),0},
{"P_ADJUST_IR2IN",P_ADJUST_IR2IN,L_D,(ValueDataUnion)(0L),0},
{"P_ALIGN_ANG",P_ALIGN_ANG,L_D,(ValueDataUnion)(0L),0},
{"P_PERF_IDX",P_PERF_IDX,D_D,(ValueDataUnion)(0.0),0},
{"P_FT_DIST",P_FT_DIST,D_D,(ValueDataUnion)(0.0),0},
{"P_ROT_TURRET_VEL",P_ROT_TURRET_VEL,L_D,(ValueDataUnion)(0L),0},
{"P_MISSION_TIME_LIMIT",P_MISSION_TIME_LIMIT,D_D,
            (ValueDataUnion)(0.0),0}
};
```

Apêndice B

Blocos Funcionais do protótipo

Este apêndice apresenta a estrutura de dados estática contendo algumas informações sobre os Blocos Funcionais (*BFs*).

```
...
/* Funcoes de processamento da decisao  */
{"function_Stop",function_Stop,0,
(int []){-1},
(int []){A_STATE_VEL_TRANSLATION,A_STATE_VEL_STEER,A_STATE_VEL_TURRET,-1}
},
{"function_AlignSline",function_AlignSline,0,
(int []){R_STATE_CONF_STEER,R_STATE_CONF_X,R_STATE_CONF_Y,-1},
(int []){A_STATE_VEL_TRANSLATION,A_STATE_VEL_STEER,V_ALIGN,-1}
},
{"function_GoToGoal",function_GoToGoal,0,
(int []){V_DIST_FRONT,R_STATE_CONF_X,R_STATE_CONF_Y,-1},
(int []){A_STATE_VEL_TRANSLATION,A_STATE_VEL_STEER,
         V_OBSTACLE,V_DIST_PI,V_DIST_GOAL,-1}
},
{"function_AlignWall",function_AlignWall,0,
(int []){V_DIST_NEAR_IDX,V_DIST_NEAR,V_DIST_12,-1},
(int []){A_STATE_VEL_TRANSLATION,A_STATE_VEL_STEER,V_ALIGN,-1}
},
{"function_FollowWall",function_FollowWall,0,
(int []){V_DIST_FRONT,V_DIST_RIGHT,V_DIST_LEFT,V_DIST_NEAR,-1},
(int []){A_STATE_VEL_TRANSLATION,A_STATE_VEL_STEER,-1}
},
```

```
/***************** Calculo de Distancias de locais  ********************/
{"function_DistGoal",function_DistGoal,0,
(int []){R_STATE_CONF_X,R_STATE_CONF_Y,-1},
(int []){V_DIST_GOAL,-1}
},
{"function_GoalAng",function_GoalAng,0,
(int []){V_DIST_FRONT,R_STATE_CONF_X,R_STATE_CONF_Y,-1},
(int []){V_GOAL_ANG,-1}
},
{"function_LimitIDist",function_LimitIDist,0,
(int []){R_STATE_CONF_X,R_STATE_CONF_Y,-1},
(int []){V_ARRIVE,-1}
},
{"function_IDist",function_IDist,0,
(int []){R_STATE_CONF_X,R_STATE_CONF_Y,-1},
(int []){V_DIST_PI,-1}
},

/************************ Distancia de Obstaculos  ***************/
{"function_ShortedDist",function_ShortedDist,0,
(int []){
V_DIST_0,V_DIST_1,V_DIST_2,V_DIST_3,V_DIST_4,
V_DIST_5,V_DIST_6,V_DIST_7,V_DIST_8,V_DIST_9,
V_DIST_10,V_DIST_11,V_DIST_12,V_DIST_13,
V_DIST_14,V_DIST_15,-1},
(int []){V_DIST_NEAR,V_DIST_NEAR_IDX,-1}
},
{"function_FrontDist",function_FrontDist,0,
(int []){V_DIST_0,V_DIST_1,V_DIST_15,-1},
(int []){V_DIST_FRONT,-1}
},
{"function_SideDist",function_SideDist,0,
(int []){V_DIST_3,V_DIST_4,V_DIST_5,
V_DIST_11,V_DIST_12,V_DIST_13,-1},
(int []){V_DIST_LEFT,V_DIST_RIGHT,-1}
},
...
```

Apêndice C

Parâmetros de Controle do protótipo

A seqüencia abaixo corresponde aos parâmetros de controle utilizados no protótipo. Existem quatro (4) configurações automáticas de valores de valores. Esta configurações são referenciadas no arquivo que descreve o Grafo de Controle Adaptativo que pode ser visto no apêndice F.

```
4
CONFIR
CONFIRSN
CONFIRSNT
CONFIRSNMP
P_LIMIT_TIME          DEFAULT   300000
P_GOAL_X              DEFAULT     2990
P_GOAL_Y              DEFAULT     -310
P_BEST_DIST           DEFAULT        0
P_BEST_X              DEFAULT        0
P_BEST_Y              DEFAULT        0
P_DIST_PI             DEFAULT      500
P_I_X                 DEFAULT        0
P_I_Y                 DEFAULT        0
P_MAX_ANG_ERR         DEFAULT        5
P_ALIGN_ANG           DEFAULT       10
P_MAX_DIST_ERR        DEFAULT       30
P_ADJUST_IR2IN        DEFAULT        2
P_DESIRE_DISTANCE     DEFAULT       24
P_MIN_DIST            DEFAULT       20
```

P_FT_DIST	DEFAULT	3.0
P_ROT_TURRET_VEL	DEFAULT	200
P_MIN_FRONT	CONFIR	24
P_MIN_FRONT	CONFIRSN	24
P_MIN_FRONT	CONFIRSNT	24
P_MIN_FRONT	CONFIRSNMP	24
P_MAX_STEER	CONFIR	450
P_MAX_STEER	CONFIRSN	450
P_MAX_STEER	CONFIRSNT	450
P_MAX_STEER	CONFIRSNMP	450
P_MAX_TRANS	CONFIR	240
P_MAX_TRANS	CONFIRSN	200
P_MAX_TRANS	CONFIRSNT	200
P_MAX_TRANS	CONFIRSNMP	140
P_PERF_IDX	CONFIR	1.0000
P_PERF_IDX	CONFIRSN	0.9414
P_PERF_IDX	CONFIRSNT	0.9346
P_PERF_IDX	CONFIRSNMP	0,6971
P_MISSION_TIME_LIMIT	DEFAULT	135.0

Apêndice D

Testes de transição de fase

Este é o conteúdo do arquivo que descreve os testes de transição de fases utilizados no protótipo. Estes testes são referenciados no Grafo de Controle Adaptativo do apêndice F.

conected	V_CONNECTED	NEQ	ZERO
low_bat	V_LOWBAT	NEQ	ZERO
aligned	V_ALIGN	NEQ	ZERO
colision	V_COLISION	NEQ	ZERO
sucess	V_DIST_GOAL	LST	P_MAX_DIST_ERR
limitIdist	V_DIST_PI	GRT	P_DIST_PI
arrived	V_ARRIVE	NEQ	ZERO
obstacle	V_OBSTACLE	NEQ	ZERO
changeDirection	V_CHANGE_TEST	NEQ	ZERO
alignedGoal	V_GOAL_ANG	ALST	P_ALIGN_ANG
timemout	R_STATE_TIME	GRT	P_LIMIT_TIME
missionfailure	V_COLISION	NEQ	ZERO
missionfailure	R_STATE_TIME	GRT	P_LIMIT_TIME
missionfailure	V_LOWBAT	NEQ	ZERO
stopped	R_STATE_VEL_TRANS	EQ	ZERO
stopped	R_STATE_VEL_STEER	EQ	ZERO
stopped	R_STATE_VEL_TURRET	EQ	ZERO
missiontlimit	SYS_MISSION_TIME	GRT	P_MISSION_TIME_LIMIT

FarIdist	V_DIST_PI	SUB	P_DIST_PI	GRT	P_DIST_PI
nearDist	V_DIST_GOAL	ADD	P_DESIRE_DISTANCE	LST	P_BEST_DIST
FarDist	V_DIST_GOAL	SUB	P_DIST_PI	GRT	P_DIST_PI

Apêndice E

Escalonamentos de Blocos Funcionais do protótipo

A seqüencia abaixo corresponde aos escalonamentos de Blocos Funcionais criados manualmente para o protótipo.

# nomesched	bfname	attrib	bftime	parameters
start	function_InitRobot	NORMAL	0	null
start	ACG_TRANS	TRANS	0	null
#				
stop	ACG_TRANS	TRANS	0	null
#				
stopping	function_SyncSensors	NORMAL	0	null
stopping	function_Stop	NORMAL	0	null
stopping	function_SyncActuators	NORMAL	0	null
stopping	ACG_TRANS	TRANS	0	null
#				
disconect	function_PrintMap	NORMAL	0	null
disconect	function_FreeMap	NORMAL	0	null
disconect	function_FreeRobot	NORMAL	0	null
disconect	ACG_TRANS	TRANS	0	null
#				
init	function_SyncSensors	NORMAL	0	null
init	function_Stop	NORMAL	0	null
init	function_InitMap	NORMAL	0	null
init	function_SyncActuators	NORMAL	0	null
init	ACG_TRANS	TRANS	0	null
#				

```
Goal            function_SyncSensors        NORMAL  0  null
Goal            function_Stop               NORMAL  0  null
Goal            function_SyncActuators      NORMAL  0  null
Goal            ACG_TRANS                   TRANS   0  null
#
AlignSline      function_SyncSensors        NORMAL  0  null
AlignSline      function_AlignSline         NORMAL  0  null
AlignSline      function_LowBat             NORMAL  0  null
AlignSline      function_ColisionDetect     NORMAL  0  null
AlignSline      function_DistGoal           NORMAL  0  null
AlignSline      function_CopySteerTurret    NORMAL  0  null
AlignSline      function_SyncActuators      NORMAL  0  null
AlignSline      ACG_TRANS                   TRANS   0  null
#
#                       IRs only + Adaptation
#
GoToGoal_I      function_SyncSensors        NORMAL  0  null
GoToGoal_I      function_MapDistIRs         NORMAL  0  null
GoToGoal_I      function_MapRotDist         NORMAL  0  null
GoToGoal_I      function_FrontDist          NORMAL  0  null
GoToGoal_I      function_GoToGoal           NORMAL  0  null
GoToGoal_I      function_LowBat             NORMAL  0  null
GoToGoal_I      function_ColisionDetect     NORMAL  0  null
GoToGoal_I      function_CopySteerTurret    NORMAL  0  null
GoToGoal_I      function_SyncActuators      NORMAL  0  null
GoToGoal_I      function_DecayConfidence    NORMAL  0  null
GoToGoal_I      ACG_TRANS                   TRANS   0  null
GoToGoal_I      ACG_ADAPT                   ADAPT   0  null
#
AlignWall_I     function_SyncSensors        NORMAL  0  null
AlignWall_I     function_MapDistIRs         NORMAL  0  null
AlignWall_I     function_MapRotDist         NORMAL  0  null
AlignWall_I     function_ShortedDist        NORMAL  0  null
AlignWall_I     function_AlignWall          NORMAL  0  null
AlignWall_I     function_LowBat             NORMAL  0  null
AlignWall_I     function_ColisionDetect     NORMAL  0  null
AlignWall_I     function_DistGoal           NORMAL  0  null
AlignWall_I     function_CopySteerTurret    NORMAL  0  null
```

AlignWall_I	function_SyncActuators	NORMAL	0	null
AlignWall_I	function_DecayConfidence	NORMAL	0	null
AlignWall_I	ACG_TRANS	TRANS	0	null
AlignWall_I	ACG_ADAPT	ADAPT	0	null
#				
GetAway_I	function_SyncSensors	NORMAL	0	null
GetAway_I	function_MapDistIRs	NORMAL	0	null
GetAway_I	function_MapRotDist	NORMAL	0	null
GetAway_I	function_FrontDist	NORMAL	0	null
GetAway_I	function_SideDist	NORMAL	0	null
GetAway_I	function_ShortedDist	NORMAL	0	null
GetAway_I	function_FollowWall	NORMAL	0	null
GetAway_I	function_IDist	NORMAL	0	null
GetAway_I	function_BestDistance	NORMAL	0	null
GetAway_I	function_InitTestDir	NORMAL	0	null
GetAway_I	function_GoalAng	NORMAL	0	null
GetAway_I	function_LowBat	NORMAL	0	null
GetAway_I	function_ColisionDetect	NORMAL	0	null
GetAway_I	function_CopySteerTurret	NORMAL	0	null
GetAway_I	function_SyncActuators	NORMAL	0	null
GetAway_I	function_DecayConfidence	NORMAL	0	null
GetAway_I	ACG_TRANS	TRANS	0	null
GetAway_I	ACG_ADAPT	ADAPT	0	null
#				
GetBack_I	function_SyncSensors	NORMAL	0	null
GetBack_I	function_MapDistIRs	NORMAL	0	null
GetBack_I	function_MapRotDist	NORMAL	0	null
GetBack_I	function_FrontDist	NORMAL	0	null
GetBack_I	function_SideDist	NORMAL	0	null
GetBack_I	function_ShortedDist	NORMAL	0	null
GetBack_I	function_FollowWall	NORMAL	0	null
GetBack_I	function_IDist	NORMAL	0	null
GetBack_I	function_DistGoal	NORMAL	0	null
GetBack_I	function_TestDir	NORMAL	0	null
GetBack_I	function_GoalAng	NORMAL	0	null
GetBack_I	function_LowBat	NORMAL	0	null
GetBack_I	function_ColisionDetect	NORMAL	0	null
GetBack_I	function_CopySteerTurret	NORMAL	0	null

GetBack_I	function_SyncActuators	NORMAL	0	null
GetBack_I	function_DecayConfidence	NORMAL	0	null
GetBack_I	ACG_TRANS	TRANS	0	null
GetAway_I	ACG_ADAPT	ADAPT	0	null
#				
#	IRs + Sonars + Adaptation			
#				
GoToGoal_IS	function_SyncSensors	NORMAL	0	null
GoToGoal_IS	function_MapDistIRs	NORMAL	0	null
GoToGoal_IS	function_MapDistSonares	NORMAL	0	null
GoToGoal_IS	function_MapRotDist	NORMAL	0	null
GoToGoal_IS	function_FrontDist	NORMAL	0	null
GoToGoal_IS	function_GoToGoal	NORMAL	0	null
GoToGoal_IS	function_LowBat	NORMAL	0	null
GoToGoal_IS	function_ColisionDetect	NORMAL	0	null
GoToGoal_IS	function_CopySteerTurret	NORMAL	0	null
GoToGoal_IS	function_SyncActuators	NORMAL	0	null
GoToGoal_IS	function_DecayConfidence	NORMAL	0	null
GoToGoal_IS	ACG_TRANS	TRANS	0	null
GoToGoal_IS	ACG_ADAPT	ADAPT	0	null
#				
AlignWall_IS	function_SyncSensors	NORMAL	0	null
AlignWall_IS	function_MapDistIRs	NORMAL	0	null
AlignWall_IS	function_MapDistSonares	NORMAL	0	null
AlignWall_IS	function_MapRotDist	NORMAL	0	null
AlignWall_IS	function_ShortedDist	NORMAL	0	null
AlignWall_IS	function_AlignWall	NORMAL	0	null
AlignWall_IS	function_LowBat	NORMAL	0	null
AlignWall_IS	function_ColisionDetect	NORMAL	0	null
AlignWall_IS	function_DistGoal	NORMAL	0	null
AlignWall_IS	function_CopySteerTurret	NORMAL	0	null
AlignWall_IS	function_SyncActuators	NORMAL	0	null
AlignWall_IS	function_DecayConfidence	NORMAL	0	null
AlignWall_IS	ACG_TRANS	TRANS	0	null
AlignWall_IS	ACG_ADAPT	ADAPT	0	null
#				
GetAway_IS	function_SyncSensors	NORMAL	0	null
GetAway_IS	function_MapDistIRs	NORMAL	0	null

GetAway_IS	function_MapDistSonares	NORMAL	0	null
GetAway_IS	function_MapRotDist	NORMAL	0	null
GetAway_IS	function_FrontDist	NORMAL	0	null
GetAway_IS	function_SideDist	NORMAL	0	null
GetAway_IS	function_ShortedDist	NORMAL	0	null
GetAway_IS	function_FollowWall	NORMAL	0	null
GetAway_IS	function_IDist	NORMAL	0	null
GetAway_IS	function_BestDistance	NORMAL	0	null
GetAway_IS	function_InitTestDir	NORMAL	0	null
GetAway_IS	function_GoalAng	NORMAL	0	null
GetAway_IS	function_LowBat	NORMAL	0	null
GetAway_IS	function_ColisionDetect	NORMAL	0	null
GetAway_IS	function_CopySteerTurret	NORMAL	0	null
GetAway_IS	function_SyncActuators	NORMAL	0	null
GetAway_IS	function_DecayConfidence	NORMAL	0	null
GetAway_IS	ACG_TRANS	TRANS	0	null
GetAway_IS	ACG_ADAPT	ADAPT	0	null
#				
GetBack_IS	function_SyncSensors	NORMAL	0	null
GetBack_IS	function_MapDistIRs	NORMAL	0	null
GetBack_IS	function_MapDistSonares	NORMAL	0	null
GetBack_IS	function_MapRotDist	NORMAL	0	null
GetBack_IS	function_FrontDist	NORMAL	0	null
GetBack_IS	function_SideDist	NORMAL	0	null
GetBack_IS	function_ShortedDist	NORMAL	0	null
GetBack_IS	function_FollowWall	NORMAL	0	null
GetBack_IS	function_IDist	NORMAL	0	null
GetBack_IS	function_DistGoal	NORMAL	0	null
GetBack_IS	function_TestDir	NORMAL	0	null
GetBack_IS	function_GoalAng	NORMAL	0	null
GetBack_IS	function_LowBat	NORMAL	0	null
GetBack_IS	function_ColisionDetect	NORMAL	0	null
GetBack_IS	function_CopySteerTurret	NORMAL	0	null
GetBack_IS	function_SyncActuators	NORMAL	0	null
GetBack_IS	function_DecayConfidence	NORMAL	0	null
GetBack_IS	ACG_TRANS	TRANS	0	null
GetBack_IS	ACG_ADAPT	ADAPT	0	null
#				

```
#                    IRs + Sonars +Fault Detect Test + Adaptation
#
GoToGoal_IST   function_SyncSensors          NORMAL   0   null
GoToGoal_IST   function_MapDistIRs           NORMAL   0   null
GoToGoal_IST   function_MapDistSonares       NORMAL   0   null
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_0—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_1—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_2—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_3—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_4—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_5—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_6—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_7—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_8—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_9—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_10—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_11—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_12—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_13—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_14—P_FT_DIST
GoToGoal_IST   function_ft_diffvalue_SNIR    FDED     0   R_DIST_15—P_FT_DIST
GoToGoal_IST   function_MapRotDist           NORMAL   0   null
GoToGoal_IST   function_FrontDist            NORMAL   0   null
GoToGoal_IST   function_GoToGoal             NORMAL   0   null
GoToGoal_IST   function_LowBat               NORMAL   0   null
GoToGoal_IST   function_ColisionDetect       NORMAL   0   null
GoToGoal_IST   function_CopySteerTurret      NORMAL   0   null
GoToGoal_IST   function_SyncActuators        NORMAL   0   null
GoToGoal_IST   ACG_TRANS                     TRANS    0   null
GoToGoal_IST   ACG_ADAPT                     ADAPT    0   null
#
AlignWall_IST  function_SyncSensors          NORMAL   0   null
AlignWall_IST  function_MapDistIRs           NORMAL   0   null
AlignWall_IST  function_MapDistSonares       NORMAL   0   null
AlignWall_IST  function_ft_diffvalue_SNIR    FDED     0   R_DIST_0—P_FT_DIST
AlignWall_IST  function_ft_diffvalue_SNIR    FDED     0   R_DIST_1—P_FT_DIST
AlignWall_IST  function_ft_diffvalue_SNIR    FDED     0   R_DIST_2—P_FT_DIST
AlignWall_IST  function_ft_diffvalue_SNIR    FDED     0   R_DIST_3—P_FT_DIST
```

AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_4—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_5—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_6—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_7—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_8—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_9—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_10—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_11—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_12—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_13—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_14—P_FT_DIST
AlignWall_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_15—P_FT_DIST
AlignWall_IST	function_MapRotDist	NORMAL	0	null
AlignWall_IST	function_ShortedDist	NORMAL	0	null
AlignWall_IST	function_AlignWall	NORMAL	0	null
AlignWall_IST	function_LowBat	NORMAL	0	null
AlignWall_IST	function_ColisionDetect	NORMAL	0	null
AlignWall_IST	function_DistGoal	NORMAL	0	null
AlignWall_IST	function_CopySteerTurret	NORMAL	0	null
AlignWall_IST	function_SyncActuators	NORMAL	0	null
AlignWall_IST	ACG_TRANS	TRANS	0	null
AlignWall_IST	ACG_ADAPT	ADAPT	0	null
#				
GetAway_IST	function_SyncSensors	NORMAL	0	null
GetAway_IST	function_MapDistIRs	NORMAL	0	null
GetAway_IST	function_MapDistSonares	NORMAL	0	null
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_0—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_1—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_2—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_3—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_4—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_5—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_6—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_7—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_8—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_9—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_10—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_11—P_FT_DIST

GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_12—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_13—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_14—P_FT_DIST
GetAway_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_15—P_FT_DIST
GetAway_IST	function_MapRotDist	NORMAL	0	null
GetAway_IST	function_FrontDist	NORMAL	0	null
GetAway_IST	function_SideDist	NORMAL	0	null
GetAway_IST	function_ShortedDist	NORMAL	0	null
GetAway_IST	function_FollowWall	NORMAL	0	null
GetAway_IST	function_IDist	NORMAL	0	null
GetAway_IST	function_BestDistance	NORMAL	0	null
GetAway_IST	function_InitTestDir	NORMAL	0	null
GetAway_IST	function_GoalAng	NORMAL	0	null
GetAway_IST	function_LowBat	NORMAL	0	null
GetAway_IST	function_ColisionDetect	NORMAL	0	null
GetAway_IST	function_CopySteerTurret	NORMAL	0	null
GetAway_IST	function_SyncActuators	NORMAL	0	null
GetAway_IST	ACG_TRANS	TRANS	0	null
GetAway_IST	ACG_ADAPT	ADAPT	0	null
#				
GetBack_IST	function_SyncSensors	NORMAL	0	null
GetBack_IST	function_MapDistIRs	NORMAL	0	null
GetBack_IST	function_MapDistSonares	NORMAL	0	null
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_0—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_1—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_2—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_3—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_4—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_5—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_6—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_7—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_8—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_9—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_10—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_11—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_12—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_13—P_FT_DIST
GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_14—P_FT_DIST

GetBack_IST	function_ft_diffvalue_SNIR	FDED	0	R_DIST_15—P_FT_DIST
GetBack_IST	function_MapRotDist	NORMAL	0	null
GetBack_IST	function_FrontDist	NORMAL	0	null
GetBack_IST	function_SideDist	NORMAL	0	null
GetBack_IST	function_ShortedDist	NORMAL	0	null
GetBack_IST	function_FollowWall	NORMAL	0	null
GetBack_IST	function_IDist	NORMAL	0	null
GetBack_IST	function_DistGoal	NORMAL	0	null
GetBack_IST	function_TestDir	NORMAL	0	null
GetBack_IST	function_GoalAng	NORMAL	0	null
GetBack_IST	function_LowBat	NORMAL	0	null
GetBack_IST	function_ColisionDetect	NORMAL	0	null
GetBack_IST	function_CopySteerTurret	NORMAL	0	null
GetBack_IST	function_SyncActuators	NORMAL	0	null
GetBack_IST	ACG_TRANS	TRANS	0	null
GetBack_IST	ACG_ADAPT	ADAPT	0	null
#				
#	IRs + Sonars + Map+ Fault Detect + Adaptation			
#				
GoToGoal_ISMT	function_SyncSensors	NORMAL	0	null
GoToGoal_ISMT	function_MapDistIRs	NORMAL	0	null
GoToGoal_ISMT	function_MapDistSonares	NORMAL	0	null
GoToGoal_ISMT	function_GetPosSensors	NORMAL	0	null
GoToGoal_ISMT	function_Process_Map	NORMAL	0	null
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_0—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_1—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_2—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_3—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_4—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_5—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_6—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_7—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_8—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_9—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_10—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_11—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_12—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_13—P_FT_DIST

GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_14—P_FT_DIST
GoToGoal_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_15—P_FT_DIST
GoToGoal_ISMT	function_MapRotDist	NORMAL	0	null
GoToGoal_ISMT	function_FrontDist	NORMAL	0	null
GoToGoal_ISMT	function_GoToGoal	NORMAL	0	null
GoToGoal_ISMT	function_LowBat	NORMAL	0	null
GoToGoal_ISMT	function_ColisionDetect	NORMAL	0	null
GoToGoal_ISMT	function_RotateTurret	NORMAL	0	null
GoToGoal_ISMT	function_SyncActuators	NORMAL	0	null
GoToGoal_ISMT	ACG_TRANS	TRANS	0	null
GoToGoal_ISMT	ACG_ADAPT	ADAPT	0	null
#				
AlignWall_ISMT	function_SyncSensors	NORMAL	0	null
AlignWall_ISMT	function_MapDistIRs	NORMAL	0	null
AlignWall_ISMT	function_MapDistSonares	NORMAL	0	null
AlignWall_ISMT	function_GetPosSensors	NORMAL	0	null
AlignWall_ISMT	function_Process_Map	NORMAL	0	null
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_0—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_1—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_2—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_3—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_4—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_5—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_6—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_7—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_8—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_9—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_10—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_11—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_12—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_13—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_14—P_FT_DIST
AlignWall_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_15—P_FT_DIST
AlignWall_ISMT	function_MapRotDist	NORMAL	0	null
AlignWall_ISMT	function_ShortedDist	NORMAL	0	null
AlignWall_ISMT	function_AlignWall	NORMAL	0	null
AlignWall_ISMT	function_LowBat	NORMAL	0	null
AlignWall_ISMT	function_ColisionDetect	NORMAL	0	null

AlignWall_ISMT	function_DistGoal	NORMAL	0	null
AlignWall_ISMT	function_RotateTurret	NORMAL	0	null
AlignWall_ISMT	function_SyncActuators	NORMAL	0	null
AlignWall_ISMT	ACG_TRANS	TRANS	0	null
AlignWall_ISMT	ACG_ADAPT	ADAPT	0	null
#				
GetAway_ISMT	function_SyncSensors	NORMAL	0	null
GetAway_ISMT	function_MapDistIRs	NORMAL	0	null
GetAway_ISMT	function_MapDistSonares	NORMAL	0	null
GetAway_ISMT	function_GetPosSensors	NORMAL	0	null
GetAway_ISMT	function_Process_Map	NORMAL	0	null
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_0—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_1—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_2—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_3—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_4—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_5—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_6—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_7—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_8—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_9—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_10—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_11—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_12—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_13—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_14—P_FT_DIST
GetAway_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_15—P_FT_DIST
GetAway_ISMT	function_MapRotDist	NORMAL	0	null
GetAway_ISMT	function_FrontDist	NORMAL	0	null
GetAway_ISMT	function_SideDist	NORMAL	0	null
GetAway_ISMT	function_ShortedDist	NORMAL	0	null
GetAway_ISMT	function_FollowWall	NORMAL	0	null
GetAway_ISMT	function_IDist	NORMAL	0	null
GetAway_ISMT	function_BestDistance	NORMAL	0	null
GetAway_ISMT	function_InitTestDir	NORMAL	0	null
GetAway_ISMT	function_GoalAng	NORMAL	0	null
GetAway_ISMT	function_LowBat	NORMAL	0	null
GetAway_ISMT	function_ColisionDetect	NORMAL	0	null

GetAway_ISMT	function_RotateTurret	NORMAL	0	null
GetAway_ISMT	function_SyncActuators	NORMAL	0	null
GetAway_ISMT	ACG_TRANS	TRANS	0	null
GetAway_ISMT	ACG_ADAPT	ADAPT	0	null
#				
GetBack_ISMT	function_SyncSensors	NORMAL	0	null
GetBack_ISMT	function_MapDistIRs	NORMAL	0	null
GetBack_ISMT	function_MapDistSonares	NORMAL	0	null
GetBack_ISMT	function_GetPosSensors	NORMAL	0	null
GetBack_ISMT	function_Process_Map	NORMAL	0	null
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_0—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_1—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_2—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_3—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_4—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_5—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_6—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_7—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_8—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_9—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_10—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_11—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_12—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_13—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_14—P_FT_DIST
GetBack_ISMT	function_ft_diffvalue_SNIR	FDED	0	R_DIST_15—P_FT_DIST
GetBack_ISMT	function_MapRotDist	NORMAL	0	null
GetBack_ISMT	function_FrontDist	NORMAL	0	null
GetBack_ISMT	function_SideDist	NORMAL	0	null
GetBack_ISMT	function_ShortedDist	NORMAL	0	null
GetBack_ISMT	function_FollowWall	NORMAL	0	null
GetBack_ISMT	function_IDist	NORMAL	0	null
GetBack_ISMT	function_DistGoal	NORMAL	0	null
GetBack_ISMT	function_TestDir	NORMAL	0	null
GetBack_ISMT	function_GoalAng	NORMAL	0	null
GetBack_ISMT	function_LowBat	NORMAL	0	null
GetBack_ISMT	function_ColisionDetect	NORMAL	0	null
GetBack_ISMT	function_RotateTurret	NORMAL	0	null

GetBack_ISMT	function_SyncActuators	NORMAL	0	null
GetBack_ISMT	ACG_TRANS	TRANS	0	null
GetBack_ISMT	ACG_ADAPT	ADAPT	0	null

Apêndice F

Grafo de Controle Adaptativo do protótipo

Este é o conteúdo do arquivo que descreve o Grafo de Controle Adaptativo utilizado no protótipo desenvolvido.

#	Nodes			
#n	nodename	schedname	confname	aftid
#				
n	Stop	stop	CONFIR	IR
n	Start	start	CONFIR	IR
n	MissionInit	init	CONFIR	IR
n	MissionFailure	stopping	CONFIR	IR
n	Disconect	disconect	CONFIR	IR
n	MissionSucess	stopping	CONFIR	IR
#				
#		IR		
#				
n	AlignSline_I	AlignSline	CONFIR	IR
n	GoToGoal_I	GoToGoal_I	CONFIR	IR
n	AlignWall_I	AlignWall_I	CONFIR	IR
n	GetAway_I	GetAway_I	CONFIR	IR
n	GetBack_I	GetBack_I	CONFIR	IR
#				
#		IR + Sonars		
#				
n	AlignSline_IS	AlignSline	CONFIRSN	SNIR
n	GoToGoal_IS	GoToGoal_IS	CONFIRSN	SNIR

```
n    AlignWall_IS          AlignWall_IS          CONFIRSN      SNIR
n    GetAway_IS            GetAway_IS            CONFIRSN      SNIR
n    GetBack_IS            GetBack_IS            CONFIRSN      SNIR
#
#                          IR + Sonars + Tests
#
n    AlignSline_IST        AlignSline            CONFIRSNT     SNIRT
n    GoToGoal_IST          GoToGoal_IST          CONFIRSNT     SNIRT
n    AlignWall_IST         AlignWall_IST         CONFIRSNT     SNIRT
n    GetAway_IST           GetAway_IST           CONFIRSNT     SNIRT
n    GetBack_IST           GetBack_IST           CONFIRSNT     SNIRT
#
#                          IR + Sonars + Map + Tests
#
n    AlignSline_ISMT       AlignSline            CONFIRSNMP    SNIRMPT
n    GoToGoal_ISMT         GoToGoal_ISMT         CONFIRSNMP    SNIRMPT
n    AlignWall_ISMT        AlignWall_ISMT        CONFIRSNMP    SNIRMPT
n    GetAway_ISMT          GetAway_ISMT          CONFIRSNMP    SNIRMPT
n    GetBack_ISMT          GetBack_ISMT          CONFIRSNMP    SNIRMPT
#
#    Edges
#e   soucenode              destnode              edgetype      testid
#
e    Start                 MissionInit           TRANS         conected
e    Start                 Stop                  TRANS         TRUE
e    MissionInit           AlignSline_IST        TRANS         TRUE
e    MissionSucess         Disconect             TRANS         stopped
e    MissionFailure        Disconect             TRANS         stopped
e    Disconect             Stop                  TRANS         TRUE
#
e    AlignSline_I          GoToGoal_I            TRANS         aligned
e    AlignSline_I          MissionSucess         TRANS         sucess
e    AlignSline_I          MissionFailure        TRANS         colision
e    AlignSline_I          MissionFailure        TRANS         missiontlimit
#
e    AlignSline_IS         GoToGoal_IS           TRANS         aligned
e    AlignSline_IS         MissionSucess         TRANS         sucess
e    AlignSline_IS         MissionFailure        TRANS         colision
```

```
e   AlignSline_IS       MissionFailure      TRANS   missiontlimit
#
e   AlignSline_IST      GoToGoal_IST        TRANS   aligned
e   AlignSline_IST      MissionSucess       TRANS   sucess
e   AlignSline_IST      MissionFailure      TRANS   colision
e   AlignSline_IST      MissionFailure      TRANS   missiontlimit
#
e   AlignSline_ISMT     GoToGoal_ISMT       TRANS   aligned
e   AlignSline_ISMT     MissionSucess       TRANS   sucess
e   AlignSline_ISMT     MissionFailure      TRANS   colision
e   AlignSline_ISMT     MissionFailure      TRANS   missiontlimit
#
e   GoToGoal_I          AlignWall_I         TRANS   obstacle
e   GoToGoal_I          MissionSucess       TRANS   sucess
e   GoToGoal_I          MissionFailure      TRANS   colision
e   GoToGoal_I          MissionFailure      TRANS   missiontlimit
e   GoToGoal_I          GoToGoal_IS         ADAPT   TRUE
#
e   GoToGoal_IS         AlignWall_IS        TRANS   obstacle
e   GoToGoal_IS         MissionSucess       TRANS   sucess
e   GoToGoal_IS         MissionFailure      TRANS   colision
e   GoToGoal_IS         MissionFailure      TRANS   missiontlimit
e   GoToGoal_IS         GoToGoal_I          ADAPT   TRUE
e   GoToGoal_IS         GoToGoal_IST        ADAPT   TRUE
#
e   GoToGoal_IST        AlignWall_IST       TRANS   obstacle
e   GoToGoal_IST        MissionSucess       TRANS   sucess
e   GoToGoal_IST        MissionFailure      TRANS   colision
e   GoToGoal_IST        MissionFailure      TRANS   missiontlimit
e   GoToGoal_IST        GoToGoal_IS         ADAPT   TRUE
e   GoToGoal_IST        GoToGoal_ISMT       ADAPT   TRUE
#
e   GoToGoal_ISMT       AlignWall_ISMT      TRANS   obstacle
e   GoToGoal_ISMT       MissionSucess       TRANS   sucess
e   GoToGoal_ISMT       MissionFailure      TRANS   colision
e   GoToGoal_ISMT       MissionFailure      TRANS   missiontlimit
e   GoToGoal_ISMT       GoToGoal_IST        ADAPT   TRUE
#
```

```
e   AlignWall_I      GetAway_I       TRANS   aligned
e   AlignWall_I      MissionSucess   TRANS   sucess
e   AlignWall_I      MissionFailure  TRANS   colision
e   AlignWall_I      MissionFailure  TRANS   missiontlimit
e   AlignWall_I      AlignWall_IS    ADAPT   TRUE
#
e   AlignWall_IS     GetAway_IS      TRANS   aligned
e   AlignWall_IS     MissionSucess   TRANS   sucess
e   AlignWall_IS     MissionFailure  TRANS   colision
e   AlignWall_IS     MissionFailure  TRANS   missiontlimit
e   AlignWall_IS     AlignWall_I     ADAPT   TRUE
e   AlignWall_IS     AlignWall_IST   ADAPT   TRUE
#
e   AlignWall_IST    GetAway_IST     TRANS   aligned
e   AlignWall_IST    MissionSucess   TRANS   sucess
e   AlignWall_IST    MissionFailure  TRANS   colision
e   AlignWall_IST    MissionFailure  TRANS   missiontlimit
e   AlignWall_IST    AlignWall_IS    ADAPT   TRUE
e   AlignWall_IST    AlignWall_ISMT  ADAPT   TRUE
#
e   AlignWall_ISMT   GetAway_ISMT    TRANS   aligned
e   AlignWall_ISMT   MissionSucess   TRANS   sucess
e   AlignWall_ISMT   MissionFailure  TRANS   colision
e   AlignWall_ISMT   MissionFailure  TRANS   missiontlimit
e   AlignWall_ISMT   AlignWall_IST   ADAPT   TRUE
#
e   GetAway_I       GetBack_I       TRANS   limitIdist
e   GetAway_I       MissionSucess   TRANS   sucess
e   GetAway_I       MissionFailure  TRANS   colision
e   GetAway_I       MissionFailure  TRANS   missiontlimit
e   GetAway_I       GetAway_IS      ADAPT   TRUE
#
e   GetAway_IS      GetBack_IS      TRANS   limitIdist
e   GetAway_IS      MissionSucess   TRANS   sucess
e   GetAway_IS      MissionFailure  TRANS   colision
e   GetAway_IS      MissionFailure  TRANS   missiontlimit
e   GetAway_IS      GetAway_I       ADAPT   TRUE
e   GetAway_IS      GetAway_IST     ADAPT   TRUE
```

```
#
e   GetAway_IST     GetBack_IST     TRANS   limitIdist
e   GetAway_IST     MissionSucess   TRANS   sucess
e   GetAway_IST     MissionFailure  TRANS   colision
e   GetAway_IST     MissionFailure  TRANS   missiontlimit
e   GetAway_IST     GetAway_IS      ADAPT   TRUE
e   GetAway_IST     GetAway_ISMT    ADAPT   TRUE
#
e   GetAway_ISMT    GetBack_ISMT    TRANS   limitIdist
e   GetAway_ISMT    MissionSucess   TRANS   sucess
e   GetAway_ISMT    MissionFailure  TRANS   colision
e   GetAway_ISMT    MissionFailure  TRANS   missiontlimit
e   GetAway_ISMT    GetAway_IST     ADAPT   TRUE
#
e   GetBack_I       AlignSline_I    TRANS   changeDirection
e   GetBack_I       AlignSline_I    TRANS   alignedGoal
e   GetBack_I       AlignSline_I    TRANS   FarIdist
e   GetBack_I       MissionSucess   TRANS   sucess
e   GetBack_I       MissionFailure  TRANS   colision
e   GetBack_I       MissionFailure  TRANS   missiontlimit
e   GetBack_I       GetBack_IS      ADAPT   TRUE
#
e   GetBack_IS      AlignSline_IS   TRANS   changeDirection
e   GetBack_IS      AlignSline_IS   TRANS   alignedGoal
e   GetBack_IS      AlignSline_IS   TRANS   FarIdist
e   GetBack_IS      MissionSucess   TRANS   sucess
e   GetBack_IS      MissionFailure  TRANS   colision
e   GetBack_IS      MissionFailure  TRANS   missiontlimit
e   GetBack_IS      GetBack_I       ADAPT   TRUE
e   GetBack_IS      GetBack_IST     ADAPT   TRUE
#
e   GetBack_IST     AlignSline_IST  TRANS   changeDirection
e   GetBack_IST     AlignSline_IST  TRANS   alignedGoal
e   GetBack_IST     AlignSline_IST  TRANS   FarIdist
e   GetBack_IST     MissionSucess   TRANS   sucess
e   GetBack_IST     MissionFailure  TRANS   colision
e   GetBack_IST     MissionFailure  TRANS   missiontlimit
e   GetBack_IST     GetBack_IS      ADAPT   TRUE
```

```
e    GetBack_IST      GetBack_ISMT      ADAPT    TRUE
#
e    GetBack_ISMT     AlignSline_ISMT   TRANS    changeDirection
e    GetBack_ISMT     AlignSline_ISMT   TRANS    alignedGoal
e    GetBack_ISMT     AlignSline_ISMT   TRANS    FarIdist
e    GetBack_ISMT     MissionSucess     TRANS    sucess
e    GetBack_ISMT     MissionFailure    TRANS    colision
e    GetBack_ISMT     MissionFailure    TRANS    missiontlimit
e    GetBack_ISMT     GetBack_IST       ADAPT    TRUE
#
```

Apêndice G

Arquivo de saída do comando *gprof*

Este é o conteúdo do arquivo de saída do comando do *gprof* do *Linux*, apresentando a procentagem de tempo consumida por cada função do programa. O resultado mostrado corresponde a uma execução da configuração IRSNMT.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
36.83	101.44	101.44	5926304	0.02	0.02	AddSensorLine
31.86	189.19	87.75	2963152	0.03	0.03	ReadSensorLineConf
2.84	197.00	7.81	25858539	0.00	0.00	GetInsEDItem
2.49	203.86	6.86				read_reply_from_socket
2.36	210.35	6.49	2963152	0.00	0.00	function_ft_diffvalue_SNIR
2.26	216.58	6.23	5926304	0.00	0.02	AddSensorToMap
1.94	221.93	5.35	8889456	0.00	0.00	GetRobotConfig
1.85	227.02	5.09	25858539	0.00	0.00	SelectEDItem
1.51	231.17	4.15	5926304	0.00	0.00	update_confidence_disc
1.47	235.21	4.04	26051583	0.00	0.00	GetConfidence
0.94	237.80	2.59	28256922	0.00	0.00	Copy_Item_Attib
0.89	240.25	2.45	22307190	0.00	0.00	L_GETED
0.88	242.67	2.42	23132972	0.00	0.00	TestGETED
0.78	244.81	2.14	2963152	0.00	0.03	GetMapSensorConf
0.75	246.88	2.07	1635245	0.00	0.00	MarkPointOb
0.70	248.82	1.94	6614814	0.00	0.04	execute_function
0.53	250.29	1.47				posDataProcess
0.45	251.54	1.25	2126400	0.00	0.00	Calc_R_DIST_CONF
0.44	252.75	1.21	54383858	0.00	0.00	Set_debug
0.44	253.95	1.20	185197	0.01	0.01	TranslateMap
0.41	255.08	1.13	185197	0.01	1.15	function_Process_Map

0.37	256.09	1.01	16249751	0.00	0.00	SetItemAttrib
0.36	257.09	1.00	23132972	0.00	0.00	adjust_conf_ed_calc
0.36	258.08	0.99	15230445	0.00	0.00	L_SETED
0.35	259.04	0.96	14815760	0.00	0.00	SetConfDependence
0.32	259.91	0.87	1010132	0.00	0.00	EvalTest
0.31	260.75	0.84				process_state_reply
0.29	261.55	0.80	1009972	0.00	0.00	eval_test
0.26	262.26	0.71	8889456	0.00	0.00	L_SETED_CF
0.25	262.95	0.69	9801832	0.00	0.00	SetItemAttribCf
0.25	263.63	0.68	3240114	0.00	0.00	reset_conf_ed_calc
0.23	264.27	0.64	5199303	0.00	0.00	TestGETPAR
0.23	264.90	0.63	228094	0.00	1.15	execute_Cycle_schedule
0.22	265.50	0.60	227934	0.00	0.04	function_SyncSensors
0.21	266.07	0.57	2236151	0.00	0.00	L_GETPAR
0.20	266.61	0.54	11847970	0.00	0.00	double_op
0.19	267.13	0.52	185197	0.00	0.03	function_MapRotDist
0.19	267.64	0.51				vm
0.16	268.08	0.44	228094	0.00	0.01	Find_Next_Node_transition
0.14	268.47	0.39	2951452	0.00	0.00	ChangeSelectEDItem
0.14	268.86	0.39	228094	0.00	0.00	AdjustNodeTime
0.14	269.24	0.38	962390	0.00	0.00	Calc_Op_Conf
0.13	269.59	0.35				posSonarRingGet
0.12	269.92	0.33	write_request_to_socket			
0.11	270.23	0.31	185197	0.00	0.02	function_MapDistIRs
0.11	270.54	0.31				posInfraredRingGet
0.11	270.84	0.30	185197	0.00	0.02	function_MapDistSonares
0.11	271.13	0.29	687052	0.00	0.00	getdoubletime
0.10	271.41	0.28	1009972	0.00	0.00	EvalTestModel
0.09	271.65	0.24	2963152	0.00	0.00	D_GETPAR
0.08	271.87	0.22				posDataCheck
0.08	272.08	0.21	2963152	0.00	0.00	ResetConfDependence
0.07	272.27	0.19	gs			
0.06	272.43	0.16	366908	0.00	0.00	PerfIndex
0.06	272.59	0.16				voltDataProcess
0.05	272.74	0.15	80	1.88	3295.49	Global_Schedule
0.05	272.88	0.14	112060	0.00	0.01	function_FollowWall
0.05	273.01	0.13	370394	0.00	0.00	P_SETED
0.05	273.14	0.13	366908	0.00	0.00	GetProfit

0.04	273.26	0.12	5926304	0.00	0.00	CONF_GETED
0.04	273.38	0.12	339754	0.00	0.00	CalcDist
0.04	273.50	0.12	183454	0.00	0.01	Find_Better_Node_Adaptaion
0.04	273.62	0.12	23120	0.01	0.01	GetValueIdFromName
0.04	273.73	0.11	183454	0.00	0.01	ACGAdaptTest
0.04	273.83	0.10	228094	0.00	0.01	ACGTransitionTest
0.04	273.93	0.10	158851	0.00	0.01	function_ShortedDist
0.03	274.02	0.09	227934	0.00	0.00	function_SyncActuators
0.03	274.11	0.09	99360	0.00	0.00	TrimComent
0.03	274.20	0.09				voltConvert
0.03	274.28	0.08	455868	0.00	0.00	D_SETED
0.03	274.36	0.08	455388	0.00	0.00	D_GETED
0.03	274.44	0.08	228094	0.00	0.00	UL_SETED_CF
0.02	274.50	0.06	370394	0.00	0.00	P_GETED
0.02	274.56	0.06	138406	0.00	0.00	function_FrontDist
0.02	274.62	0.06	33200	0.00	0.00	GetFuncIdFromName
0.02	274.68	0.06				voltCpuGet
0.02	274.73	0.05	193044	0.00	0.00	L_SETPAR
0.02	274.78	0.05	142127	0.00	0.00	function_DistGoal
0.02	274.83	0.05	112060	0.00	0.00	function_SideDist
0.02	274.88	0.05	52599	0.00	0.00	eval_test_3op
0.01	274.92	0.04	185197	0.00	0.00	function_GetPosSensors
0.01	274.96	0.04	185197	0.00	0.00	function_RotateTurret
0.01	275.00	0.04	154557	0.00	0.00	CalcAngTo
0.01	275.04	0.04	112060	0.00	0.00	function_IDist
0.01	275.08	0.04	42497	0.00	0.01	function_AlignSline
0.01	275.12	0.04	18880	0.00	0.00	GetNodeIdFromName
0.01	275.15	0.03	228094	0.00	0.00	InitCycleTimer
0.01	275.18	0.03	227694	0.00	0.00	function_ColisionDetect
0.01	275.21	0.03	227694	0.00	0.00	function_LowBat
0.01	275.24	0.03	112060	0.00	0.00	function_GoalAng
0.01	275.26	0.02	228094	0.00	0.01	SetSystemEDs
0.01	275.28	0.02	52839	0.00	0.00	function_TestDir
0.01	275.30	0.02	46791	0.00	0.00	function_AlignWall
0.01	275.32	0.02	26346	0.00	0.01	function_GoToGoal
0.01	275.34	0.02				posTimeGet
0.01	275.36	0.02				timeDataProcess
0.00	275.37	0.01	684282	0.00	0.00	D_SETED_CF

0.00	275.38	0.01	59221	0.00	0.00	function_BestDistance
0.00	275.39	0.01	80	0.12	0.12	PrintFACGTimes
0.00	275.40	0.01	80	0.12	2.94	read_schedules
0.00	275.40	0.00	102720	0.00	0.00	InitValueItem
0.00	275.40	0.00	59221	0.00	0.00	function_InitTestDir
0.00	275.40	0.00	42497	0.00	0.00	function_CopySteerTurret
0.00	275.40	0.00	32960	0.00	0.00	SchedAttrib
0.00	275.40	0.00	32960	0.00	0.00	SchedAttribFT
0.00	275.40	0.00	11120	0.00	0.00	GetTestIdFromTestName
0.00	275.40	0.00	10240	0.00	0.00	initvalue
0.00	275.40	0.00	9440	0.00	0.00	GetETFromName
0.00	275.40	0.00	7943	0.00	0.00	printe
0.00	275.40	0.00	3920	0.00	0.00	GetSchedIdFromName
0.00	275.40	0.00	3360	0.00	0.00	GetConfIdFromName
0.00	275.40	0.00	2880	0.00	0.00	GetTestMaskFromTestDEF
0.00	275.40	0.00	2880	0.00	0.00	initFunction
0.00	275.40	0.00	2720	0.00	0.00	GetParIdFromName
0.00	275.40	0.00	2080	0.00	0.00	GetAFTIdFromName
0.00	275.40	0.00	1680	0.00	0.00	CheckTestModel
0.00	275.40	0.00	240	0.00	0.00	SetParameterConfigIndex
0.00	275.40	0.00	240	0.00	0.00	function_Stop
0.00	275.40	0.00	80	0.00	0.01	InitTestIDs
0.00	275.40	0.00	80	0.00	3.99	Read_Text_Configs
0.00	275.40	0.00	80	0.00	0.00	free_system
0.00	275.40	0.00	80	0.00	0.00	freemap
0.00	275.40	0.00	80	0.00	0.00	function_FreeMap
0.00	275.40	0.00	80	0.00	0.00	function_FreeRobot
0.00	275.40	0.00	80	0.00	0.00	function_InitMap
0.00	275.40	0.00	80	0.00	0.00	function_InitRobot
0.00	275.40	0.00	80	0.00	0.00	function_PrintMap
0.00	275.40	0.00	80	0.00	0.00	init_data_structures
0.00	275.40	0.00	80	0.00	0.00	init_functions
0.00	275.40	0.00	80	0.00	0.00	init_map
0.00	275.40	0.00	80	0.00	3.99	init_model
0.00	275.40	0.00	80	0.00	0.00	init_rand
0.00	275.40	0.00	80	0.00	0.00	print_test_conf
0.00	275.40	0.00	80	0.00	0.00	print_upd_conf
0.00	275.40	0.00	80	0.00	0.84	read_acg

0.00	275.40	0.00	80	0.00	0.00	read_configs
0.00	275.40	0.00	80	0.00	0.02	read_schedules_times
0.00	275.40	0.00	80	0.00	0.19	read_tests
0.00	275.40	0.00	80	0.00	0.01	test_conf