Douglas Pereira Pasqualin

TRATAMENTO DE FALHAS EM ROBÔ LEGO MINDSTORMS COM ARQUITETURA REATIVA

Santa Maria, RS

2009

Douglas Pereira Pasqualin

TRATAMENTO DE FALHAS EM ROBÔ LEGO MINDSTORMS COM ARQUITETURA REATIVA

Trabalho Final de Graduação apresentado ao Curso de Sistemas de Informação – Área de Ciências Tecnológicas, do Centro Universitário Franciscano, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. MSc. Guilherme Dhein

Santa Maria, RS

2009

Douglas Pereira Pasqualin

TRATAMENTO DE FALHAS EM ROBÔ LEGO MINDSTORMS COM ARQUITETURA REATIVA

Trabalho Final de Graduação apresentado ao Curso de Sistemas de Informação – Área de Ciências Tecnológicas, do Centro Universitário Franciscano, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

_____

MSc. Guilherme Dhein – Orientador (UNIFRA)

_____

Dr. Andre Zanki Cordenonsi (UFSM)

_____

MSc. Marcos Luís Cassal (UNIFRA)

Aprovado em 16 de Julho de 2009

AGRADECIMENTOS

A realização desse trabalho só foi possível com a ajuda e apoio de algumas pessoas especiais que gostaria de agradecer nesse espaço.

Em primeiro lugar agradeço minha esposa Daiane pelo apoio em todos os momentos, me confortando nos momentos difíceis, não me deixando desistir, sempre elevando a minha auto-estima e principalmente por agüentar meu mau-humor quando tudo parecia que ia dar errado. Essa conquista é nossa!

A minha família, com a qual sempre posso contar, em especial a minha mãe, que sempre me ensinou a ser humilde e honesto e correr atrás dos meus sonhos. Uma pessoa extraordinária, que apesar de tudo que já passou está sempre de bem com a vida e sempre disposta a ajudar os outros.

Ao meu orientador e amigo Guilherme, por estar sempre disposto a trocar uma idéia sobre o trabalho, mesmo em horários extraordinários como nos corredores da UNIFRA, e mesmo fora da UNIFRA. Além de revisar constantemente a parte escrita. Sem essa incrível presteza muitos dos resultados deste trabalho não teriam sido alcançados.

Aos meus colegas de trabalho, pelas dicas e críticas construtivas que ajudaram na realização deste projeto.

A todos os professores que contribuíram para a minha formação durante o curso.

Por fim, agradeço a UNIFRA pelo seu programa de assistência educacional, tornando possível o sonho da graduação para pessoas com poucas condições financeiras.

"Although I do firmly believe that the
brain is a machine,
whether this machine is a computer
is another question"

Rodney Brooks

RESUMO

Com o avanço tecnológico, o desenvolvimento de robôs que desempenhem as tarefas de forma autônoma está se tornando realidade. Assim, para garantir a autonomia, torna-se imprescindível que eles sejam capazes de desempenhar as tarefas programadas mesmo na presença de falhas. Uma técnica utilizada para encapsular detecção e tratamento de falhas em sensores é utilizar sensores virtuais. Sensores virtuais agregam informações de vários sensores reais, e podem ser reconfigurados conforme a ocorrência de alguma falha. Este trabalho possui dois objetivos principais: o primeiro objetivo é desenvolver um sensor virtual para efetuar o tratamento de falhas sensoriais. O segundo objetivo é efetuar o tratamento de falhas motoras, tentando manter os motores do robô sincronizados. Para isso, a velocidade de cada motor será tratada separadamente, compensando a velocidade individualmente quando necessário. As duas propostas serão testadas e validadas em um robô Lego Mindstorms NXT com arquitetura reativa.

Palavras-chave: Robótica. Lego Mindstorms. Tolerância a Falhas. Arquitetura *Subsumption*.

ABSTRACT

With the technological advance, the development of robots that carry out the tasks in an autonomous way is becoming reality. Thus, to guarantee the autonomy, it becomes indispensable that they are capable to carry out the programmed tasks even in the presence of failures. A technique used to encapsulate detection and treatment of faults in sensors is to use virtual sensors. Virtual sensors aggregate information of several real sensors, and they can be reconfigured as some fail occurs. This work has two main objectives: the first objective is to develop a virtual sensor to treat sensorial failures. The second objective is to make the treatment of motor failures by trying to keep the motors of robot synchronized. For that, the speed of each motor will be treated separately, compensating individually the speed when necessary. The two proposals will be tested and validated on a Lego Mindstorms NXT robot with reactive architecture.

Keywords: Robotic. Lego Mindstorms. Fault Tolerance. Subsumption Architecture.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A robótica está deixando de ser ficção cientifica, como era descrita nos livros de Isaac Asimov[1], publicados a partir da década de 50. Cada vez mais ela está inserida na vida do ser humano. Seja no auxílio de tarefas diárias como em uma fábrica de automóveis, na execução de tarefas de forma automatizada e na montagem dos veículos, ou mais recentemente no auxilio de tarefas extremamente complexas e delicadas como cirurgias médicas (TAYLOR; STOIANOVICI, 2003). A utilização de robôs na cirurgia permite uma maior precisão em cortes cirúrgicos, tornando menos dolorosa a recuperação do paciente (VALDEJÃO, 2008). A robótica permite também a execução de tarefas que até então eram impossíveis de serem desenvolvidas pelos seres humanos, como por exemplo, a exploração espacial de Marte, coordenada pela NASA através dos robôs sonda *Spirit*, *Opportunity* e o mais recente lançado ao espaço em 2007, *Phoenix.*

Pesquisas mais recentes na área já mostram um avanço significativo na interação de robôs em ambientes humanos, permitindo que robôs auxiliem as pessoas em tarefas diárias e rotineiras, ainda um grande desafio atualmente, visto que estes ambientes são totalmente dinâmicos, imprevisíveis e difíceis de modelar (EDSINGER, 2007).

Com o avanço tecnológico, o desenvolvimento de robôs que desempenhem as tarefas de forma autônoma está se tornando realidade. Assim, para garantir a autonomia, torna-se imprescindível que eles sejam capazes de desempenhar as tarefas programadas mesmo na presença de falhas, principalmente nos casos em que estejam inseridos em ambientes inóspitos a seres humanos, sendo necessário identificar as falhas o mais cedo possível antes que se tornem catastróficas (VERMA *et al*, 2004). Segundo (SOMANI; VAIDYA, 1997), um sistema tolerante a falhas é sinônimo de um sistema confiável e disponível.

Este trabalho possui dois objetivos principais: o primeiro é apresentar uma técnica para detecção e tratamento de falhas sensoriais em robôs, que consiste na programação de sensores virtuais, que são sensores abstratos que encapsulam o comportamento de sensores físicos. O segundo objetivo é efetuar o tratamento de falhas motoras, ou seja, tentar manter os motores do robô sincronizados. Para isso a velocidade de cada motor será tratada separadamente, compensando a velocidade individualmente quando necessário.

Para validação dessas propostas será utilizado um sistema robótico real, o Lego Mindstorms NXT.

---

[1] Autor de vários livros de ficção cientifica sobre o tema robótica. No livro "*I, Robot*" Asimov criou as três leis da robótica (ASIMOV, 2008). Seus livros inspiraram alguns filmes atuais sobre robôs.

2 REFERENCIAL TEÓRICO

2.1 LEGO MINDSTORMS NXT

Segundo Brian Bagnall (BAGNALL, 2007), o desenvolvimento do Mindstorms iniciou-se em 1987 no MIT *Media Laboratory*, com um projeto intitulado "Tijolo programável" (*Programmable Brick*) financiado pelo grupo LEGO. Em 1998 a primeira versão do tijolo foi lançada com o nome de RCX (Figura 1).

Figura 1 – Primeira versão do tijolo programável, denominado RCX. (LEJOS, 2009)

O RCX possuía um processador de oito bits rodando a 16MHZ e com 32kbytes de memória. Com o kit de peças que acompanhava o RCX denominado de Mindstorms *Robotics Invention System* (RIS), robôs com os mais diferentes propósitos podiam ser montados e programados. Conforme (BAGNALL, 2007), muitas faculdades e escolas adotaram a nova tecnologia como forma de ensino, e muitos estudantes o utilizaram para pesquisa e trabalhos de graduação.

A grande surpresa para a Lego foi que a comunidade de usuários avançados (denominados *hackers*) conseguiu quebrar a codificação utilizada pelo RCX para o controle do *hardware*, assim alternativas linguagens de programação surgiram para a programação do RCX. Segundo (BAGNALL, 2007) o sucesso de vendas do RCX foi graças às linguagens alternativas de programação (como por exemplo, Java e C) que surgiram após a abertura do seu código.

No ano de 2004 a Lego anunciou que estaria trabalhando no sucessor do RCX, assim em agosto de 2006, a nova geração do RCX foi lançada, denominado NXT (MINDSTORMS, 2008). O NXT permite a criação (montagem) e programação de robôs (Figura 2):

Figura 2 - Robôs montados com o kit Mindstorms NXT da Lego (MINDSTORMS, 2008).

Esse kit é formado basicamente pelo seguinte conjunto de peças: o NXT programável, sensor de toque, sensor de som, sensor de luz, sensor ultra-sônico e três motores. Além disso, esse kit contém 519 peças da linha Lego Technic (TECHNIC, 2008), tema de produtos da Lego, que possuem peças como eixos, engrenagens e motores, permitindo assim a criação de modelos complexos.

O NXT é o cérebro do robô. É um computador programável que permite que o robô desempenhe diferentes operações (Figura 3):

Figura 3 - NXT, o cérebro do robô (MINDSTORMS, 2008).

As principais características técnicas do NXT são: processador de 32bits rodando a 48 Mhz, 256Kbytes de memória *Flash* e 64Kbytes de memória *RAM*, comunicação com o PC através de USB ou *bluetooth*, auto-falante com qualidade de som de 8Khz e alimentação através de 6 pilhas tamanho AA. Possui 4 portas de entrada para a conexão dos sensores e 3 portas de saída para a conexão dos motores.

O sensor de toque (Figura 4) detecta quando está sendo pressionado por algum objeto e também quando é liberado pelo mesmo. Pode ser utilizado, por exemplo, para detectar choque com paredes ou obstáculos em ambientes.

Figura 4 - Sensor de Toque (MINDSTORMS, 2008).

O sensor de som (Figura 5) mede o volume de um som captado em decibéis (dB). Quanto mais perto a fonte de som está do robô, mais alto ele será captado. Dessa forma o robô pode ser programado para executar uma determinada tarefa conforme a intensidade do som (BAGNALL, 2007).

Figura 5 - Sensor de Som (MINDSTORMS, 2008).

O sensor de luz (Figura 6) é capaz de distinguir cores somente entre o claro e o escuro. Ele detecta a intensidade da luz em um ambiente e mede a intensidade da luz de superfícies coloridas, devolvendo um valor entre zero e 100 (zero significa preto e 100 significa branco).

Figura 6 - Sensor de Luz (MINDSTORMS, 2008).

O sensor ultra-sônico (Figura 7) permite que o robô detecte objetos. Pode ser utilizado para fazer com que o robô desvie de obstáculos, meça a distância de objetos e detecte movimentos. O sensor mede distancias entre zero e 255 centímetros, com uma precisão de aproximadamente três centímetros. O sensor ultra-sônico utiliza o mesmo principio da visão dos morcegos: mede a distância calculando o tempo que uma onda de som leva para atingir o objeto e retornar, como se fosse um eco (MINDSTORMS, 2008).

Figura 7 - Sensor Ultra-Sônico (MINDSTORMS, 2008).

O NXT possui três motores (Figura 8), cuja função primária é permitir que o robô se locomova pelo ambiente. Porém podem ser utilizados para outros fins como, por exemplo, girar um sensor ultra-sônico para detectar objetos no perímetro do robô. Cada motor possui um sensor de rotação, que mede a rotação em graus.

Figura 8 – Motor (MINDSTORMS, 2008).

Dessa forma é possível programar o robô para efetuar uma rotação de 180 graus, dando uma meia volta em torno do eixo, por exemplo. Além disso, é possível escolher a velocidade de rotação do motor, fazendo com que o robô se locomova mais rapidamente ou mais lentamente.

Além desses sensores que acompanham o kit, é possível adquirir sensores separadamente em lojas especializadas, tais como: bússola (mede o campo magnético da terra, apontando para o norte), câmera, acelerômetro, dentre outros.

2.1.1 Alternativas de Programação do NXT

A programação do NXT pode ser feita com o *software* que acompanha o kit, fabricado pela empresa LabView (LABVIEW, 2008) e denominado NXT-G. Segundo a Lego (MINDSTORMS, 2008), o NXT-G é extremamente simples de utilizar, bastando "arrastar" e "soltar" componentes na tela. O *firmware*[2] que acompanha o NXT é código aberto, por essa razão são facilmente encontrados na internet outros *firmwares* e bibliotecas que permitem a programação do NXT em diversas linguagens de programação, tais como *C*, *Java* e *.NET*. Isso permite uma maior liberdade para a programação do mesmo, podendo ser explorados ao máximo os recursos do robô, sendo inúmeras as possibilidades de criação e programação dos mesmos.

Um dos recursos mais interessantes do Mindstorms é a possibilidade de comunicação sem fio através da tecnologia *bluetooth*[3], sendo assim a limitação de memória do NXT não se torna um agravante para construção de robôs mais complexos. Toda a lógica de programação pode ficar hospedada em um computador, sendo somente enviados comandos para o robô, que podem ser comandos de ação (andar, parar, virar), ou comandos de leitura, solicitando os valores atuais dos sensores. No caso de ser enviado um comando de leitura, o computador recebe o valor da leitura, e todo o processamento necessário para a tomada da ação é efetuado pelo computador, que por sua vez envia um comando de ação para o robô (continue andando, vire 30 graus a esquerda, pare, etc).

Uma das linguagens alternativas existentes para a programação do NXT é a NXC (*Not eXactly C*) (NXC, 2008a). Como o nome já diz é uma linguagem de programação muito parecida com a sintaxe da linguagem C, que possui inclusive uma biblioteca para permitir a comunicação via *bluetooth* com o robô. O compilador do NXC transforma o código fonte escrito com a linguagem em *bytecodes*[4] que pode ser lidos pelo *firmware* do NXT (NXC, 2008b). Dessa forma não é necessário sobrescrever o *firmware* original do NXT.

Uma das ferramentas mais completas para a programação do NXT é o Robotc (ROBOTC, 2008), desenvolvido pela *Carnegie Mellon University*, que não é somente uma simples linguagem de programação e sim um ambiente completo para a construção de programas. Além disso, das linguagens pesquisadas, a Robotc é a única que possui um depurador de código, assim é possível acompanhar a execução do código passo a passo e

---

[2] Tipo de *software* que é utilizado para controlar diretamente o *hardware*.
[3] Provê uma maneira de conectar e trocar informações entre dispositivos através de uma freqüência de rádio de curto alcance.
[4] Código intermediário entre o código fonte e a aplicação final que será interpretado por uma máquina virtual.

identificar rapidamente algum erro que passou despercebido durante a programação. Os programas escritos em Robotc são totalmente baseados na linguagem C, portanto o *firmware* original do NXT deve ser sobrescrito. Uma desvantagem que dificulta a sua utilização no meio acadêmico é que o Robotc é uma ferramenta proprietária.

Existe um projeto bastante citado na literatura pesquisada que é o projeto LeJOS (LEJOS, 2009), que permite a escrita de programas através da linguagem Java. Ele possui duas formas distintas de efetuar a programação. A primeira é utilizar o LeJOS NXJ, que consiste numa máquina virtual Java capaz de interpretar os programas escritos em Java, Dessa forma é necessário sobrescrever o *firmware* original da Lego com o LeJOS NXJ. Uma das vantagens desse projeto é a possibilidade de escrever programas totalmente orientado a objetos e utilizar a API padrão do Java inclusive com o recurso de *multithreading*[5], além de ser compatível com o *bluetooth*. A segunda forma de efetuar a programação é utilizar o iCommand (parte do projeto LeJOS), que é um biblioteca Java especifica para se comunicar com o robô através do *bluetooth*, não sendo necessário sobrescrever o *firmware* do NXT.

Uma ferramenta que promete uma revolução na programação de robôs é o *Microsot Robotics Studio* (MSROBOTICS, 2008), que permite a integração e programação utilizando o *framework .NET* da própria *Microsoft*, além de possuir um simulador para que seja possível testar e efetuar simulações dos robôs em um ambiente virtual, antes de utilizá-los no mundo real (GATES, 2007).

Conforme (KIM; JEON, 2007 e 2008) o *Microsoft Robotics Studio* possui uma linguagem de programação própria (baseada no mesmo principio do *software* que acompanha o NXT, de "arrastar" e "soltar" componentes na tela), chamada de MVPL. Não é uma linguagem simples para iniciantes, porém a grande vantagem é poder utilizar um mesmo programa criado para o NXT para outro robô qualquer que seja compatível com o *Microsoft Robotics Studio*.

Porém, conforme (COELHO; ALPOIM; LOPES, 2007), o *Microsoft Robotics Studio* peca pela falta de documentação, pois como o propósito da ferramenta é a inovação, uma boa documentação é imprescindível para o aprendizado da mesma. Além disso, os autores não conseguiram ativar todos os sensores do NXT utilizando o *Microsot Robotics Studio,* ao contrário do que afirma sua documentação.

[5] Capacidade de rodar concorrentemente programas que foram divididos em partes.

2.2 ARQUITETURAS DE AGENTES

Segundo (WOOLDRIDGE, 1999, p.3), apesar de não existir uma definição aceita universalmente sobre o que é agente, um agente pode ser definido como "um sistema de computador que está *situado* em algum ambiente, e é capaz de executar ações de forma *autônoma* neste ambiente para alcançar os seus objetivos de projeto".

Conforme (SILVA JÚNIOR, 2003, p.38) "do ponto de vista da IA, o robô é um agente que interage com o ambiente executando ações baseadas nas informações oriundas de seus sensores", ou seja, um robô pode ser considerado um agente. Nesta seção as palavras "agente" e "robô" serão utilizadas como sinônimos.

Para (MAES, 1991) a arquitetura de um agente define como um problema pode ser decomposto em módulos e como esses módulos devem interagir, de forma a determinar as próximas ações do agente. Para (RUSSEL; NORVIG, 1995), a arquitetura de um robô, define como está organizado o objetivo de gerar as ações a partir da leitura dos sensores. Conclui-se que a escolha de uma boa arquitetura é fundamental para o desenvolvimento do robô, visto que ela definirá como o problema será estruturado em partes (módulos) e como os módulos se comunicarão entre si, gerando assim as próximas ações do robô.

Conforme (DHEIN, 2000) existem três tipos de arquiteturas: deliberativas, reativas e híbridas. As arquiteturas deliberativas são baseadas na representação simbólica do mundo e das ações e no raciocínio simbólico. Nas arquiteturas reativas as decisões são tomadas em tempo real, baseadas em pouca informação e regras simples que definem a ação em função de uma situação, não sendo necessária a representação simbólica. Além disso, existem as arquiteturas hibridas, nas quais se tenta unir as arquiteturas deliberativas e reativas. Na próxima seção será descrita uma arquitetura reativa bastante utilizada na programação de robôs.

2.2.1 Arquitetura *Subsumption*

Rodney Brooks questionou a abordagem que era utilizada na sua época para construção de robôs com comportamento inteligente, capazes de interagir com ambiente. A abordagem utilizada era a simbólica (deliberativa). Conforme (BROOKS, 1986 e BROOKS, 1991), a partir da leitura dos sensores do robô, construía-se um modelo interno do ambiente. Um módulo chamado de planejador ignorava o ambiente onde estava inserido o robô e tentava montar um plano de ação baseado no seu modelo interno do ambiente. Brooks

(BROOKS, 1991), afirma que esse modelo, chamado de "*Sense – Model – Plan - Act*" (SMPA) torna o robô especifico para o ambiente em que foi projetado (Figura 9).

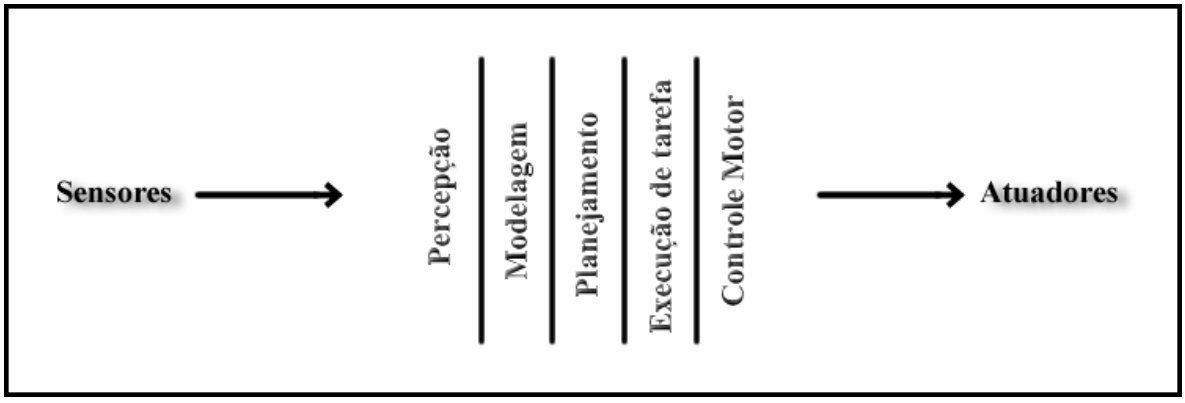

Figura 9 – Decomposição tradicional, através de módulos funcionais (BROOKS, 1986).

Outra crítica feita por Brooks à abordagem simbólica (SMPA) é que cada módulo funcional (Figura 9), não prove o comportamento do robô, mas sim a combinação deles (BROOKS, 1986 e BROOKS, 1990). Além disso, essa abordagem torna muito lenta a operação do robô, pois o gargalo está em construir o modelo interno do ambiente e só após o módulo planejador decidir a ação que deverá ser tomada (BROOKS, 1991).

A idéia principal que Brooks defende é que os robôs deixem de se referir a um modelo interno e passem a interagir diretamente com o ambiente (mundo real), pois é o melhor modelo que se pode obter, está sempre atualizado (BROOKS, 1990 e BROOKS, 1991). Além disso, Brooks argumenta que a inteligência humana se desenvolveu ao longo da evolução da espécie somente com a interação com o ambiente (BROOKS, 1990) e que a inteligência é determinada pela interação dinâmica com o mundo (BROOKS, 1991).

Baseado em todos esses argumentos, a nova abordagem sugerida por Brooks é decompor o problema em comportamentos individuais, gerando módulos que podem coexistir juntos e cooperarem para formar outros comportamentos mais complexos (BROOKS, 1986) (Figura 10). Cada fatia da Figura 10 é chamada de nível de competência, que é uma especificação informal de um comportamento desejado para o robô. Níveis mais altos representam comportamentos de maior prioridade, podendo formar assim uma hierarquia de execução. Cada nível de competência inclui como subconjunto os níveis anteriores. Além disso, cada nível de competência possui uma camada de controle (Figura 11).

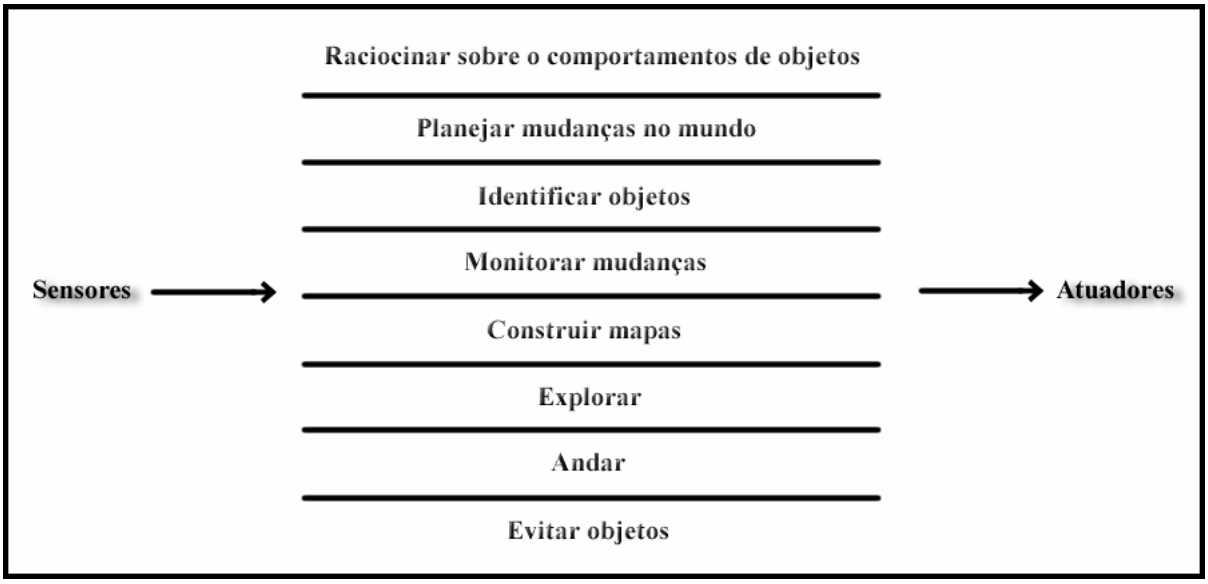

Figura 10 – Decomposição sugerida por (BROOKS, 1986), em níveis de competência

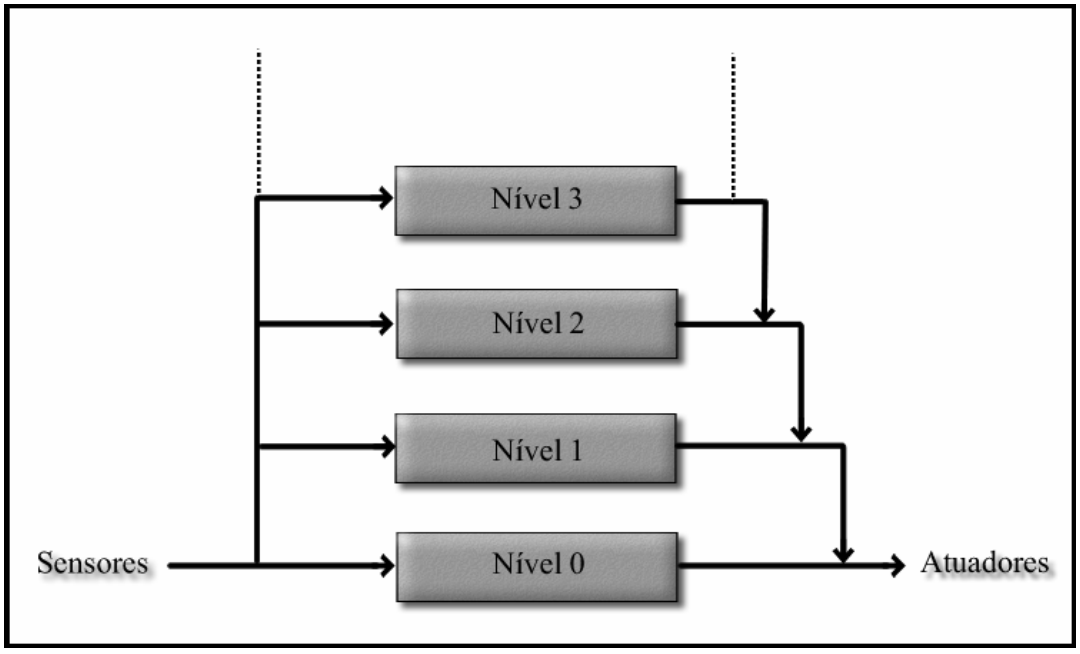

Figura 11 – Divisão em níveis de competência, proposta por (BROOKS, 1986).

Conforme (BROOKS, 1990), na abordagem simbólica, para adicionar novos comportamentos ao robô, é necessário alterar cada módulo individualmente. Nessa nova abordagem podem ser inseridos comportamentos a qualquer momento em qualquer nível, não sendo necessário reconstruir todo o sistema. Um nível mais alto pode tomar o controle suprimindo o fluxo de dados de um nível mais baixo.

A essa arquitetura Brooks denominou de *Subsumption*. Brooks recebeu muitas críticas sobre a sua abordagem, pois estava contestando quase trinta anos de estudos sobre a abordagem simbólica. Por essa razão ele possui muitos artigos publicados defendendo a sua tese como em (BROOKS, 1990), onde uma das criticas rebatidas é que com a arquitetura *Subsumption* não se pode fazer um conjunto de atividades. Brooks então defende que sua abordagem pode não ser boa para muitos problemas, mas que não se pode generalizar, nem esperar que ela resolva todos os problemas na área de robótica, da mesma forma que é injusto afirmar que não é interessante estudar a inteligência dos elefantes, somente porque eles não jogam xadrez (essa afirmação é o título do artigo (BROOKS, 1990)).

A arquitetura *Subsumption* é utilizada até hoje como principal referência para construção de robôs baseados em comportamentos e será utilizada para a elaboração deste trabalho.

2.3 TOLERÂNCIA A FALHAS EM ROBÓTICA

A tolerância à falhas em robôs autônomos é de fundamental importância, principalmente nos casos em que os robôs estão executando tarefas perigosas ou de alto risco para seres humanos, como, por exemplo, manipulação de materiais radioativos ou auxilio em operações militares. Nesses ambientes robôs devem ser capazes de continuar executando por longos períodos de tempo as tarefas programadas, mesmo na presença de falhas, caso contrário não haverá vantagem alguma em utilizar robôs nesses ambientes, pois é totalmente inviável ou às vezes até impossível enviar um humano para consertar falhas que venham a ocorrer (FERRELL, 1994), (VISINSKY; CAVALLARO; WALKER, 1995) e (VERMA *et al*, 2004);

Segundo (VERMA *et al*, 2004), a detecção de falhas em robôs é uma tarefa complexa, por uma série de fatores como: a variedade de falhas que podem acontecer é grande; sensores, atuadores e ambiente onde o robô está inserido são imprevisíveis e o poder computacional e tempo para detecção geralmente são limitados.

Segundo (SOMANI; VAIDYA, 1997), um bom projeto para se construir um sistema tolerante a falhas deve analisar primeiramente o ambiente onde o sistema (ou robô) estará inserido e determinar as falhas que poderão ocorrer e quais devem possuir tolerâncias. Em algumas situações isso pode ser uma tarefa complexa, pois é preciso ter um conhecimento prévio do sistema ou de variações normais que podem ocorrer na leitura dos sensores.

Falhas podem ser atribuídas a problemas mecânicos, eletrônicos ou nos sensores, e podem ser classificadas como falhas permanentes ou transitórias. Além disso, algumas falhas podem ocorrer no *hardware* e outras no *software* (FERRELL, 1993) e (SOMANI; VAIDYA, 1997). Outra razão de falhas pode ser atribuída a desgastes mecânicos naturais devido ao uso, e também porque raramente robôs têm conhecimento total do ambiente onde estão inseridos podendo assim se deparar com situações que não estavam programadas (VERMA *et al*, 2004).

Cynthia Ferrell (FERRELL, 1993), utilizou em seu trabalho, um robô construído com a arquitetura *Subsumption*. Ela explica que nessa arquitetura falhas em sensores podem fazer com que um comportamento seja ativado no momento errado. Dessa forma a autora argumenta que erros devem ser descobertos e isolados no nível mais baixo possível da hierarquia de comportamentos. Portanto, se o erro não for isolado no nível que ele se originou, ou seja, o erro foi repassado para os outros níveis, então os níveis mais altos devem detectar e compensar a falha. Ferrell (FERRELL, 1993) enfatiza que se isolando o erro em um nível mais baixo da hierarquia, irá maximizar a eficácia dos procedimentos e minimizar impactos no sistema.

Uma técnica utilizada para detectar erros de sensores em arquiteturas comportamentais é utilizar sensores virtuais. Sensores virtuais agregam informações de vários sensores reais, e podem ser reconfigurados conforme a ocorrência de alguma falha. Conforme (CALDAS, 2004, p.43), "os sensores virtuais são sensores abstratos que isolam os comportamentos dos sensores reais, oferecendo fontes de dados sensoriais que embutem a tolerância à falhas de sensores de forma transparente". Ou seja, os sensores virtuais são responsáveis pela detecção das falhas e pela abordagem que será utilizada para compensá-la. O programa efetua a requisição para a leitura do valor do sensor aos sensores virtuais, esses, por sua vez, lêem os valores dos sensores físicos e fazem o devido tratamento da informação quando necessário.

Conforme (VISINSKY, 1991), podem ocorrer discrepâncias entre valores lidos e valores esperados, e a causa pode ser ruído no ambiente ou entrada de dados incorretos. Um exemplo seria uma caixa que um robô deve carregar, o robô espera que a caixa esteja vazia, e o algoritmo utilizado para locomoção calculará um valor incorreto para a força que o motor deverá exercer. O controlador poderá identificar incorretamente uma falha no motor, visto que o mesmo não está se locomovendo como esperado, assim uma técnica utilizada é comparar os valores dos sensores com os valores esperados e corrigir erros introduzidos durante o processo de cálculo. Porém a desvantagem deste método é que nem sempre será possível saber previamente o valor esperado da leitura de um sensor. Caldas (2004) descreve outra situação: suponha que se tenha a informação que a leitura de um sensor varia em torno de dez unidades

(dados obtidos através de amostragem), então se dois valores lidos apresentarem diferença de mil unidades, a probabilidade de se ter uma falha nesse sensor é grande.

Para (SOMANI; VAIDYA, 1997 e CALDAS, 2004), o principio básico de tolerância a falhas é a redundância (de *hardware,* de *software*, na forma de processar uma informação, etc.). A maioria das técnicas de tolerância a falhas podem ser implementadas via *hardware* ou *software* especiais. Técnicas de *hardware* tendem a obter um melhor desempenho, porém tornam mais caro o projeto final. Técnicas de *software* são mais flexíveis, pois o *software* pode ser modificado após o sistema ter sido construído (SOMANI; VAIDYA, 1997).

Uma técnica existente para tolerância a falhas é duplicar os sensores e atuadores. Um exemplo seria utilizar mais de um sensor ultra-sônico e utilizar como referencia a média da leitura dos sensores como valor final, ou seja, um sensor virtual simples. Outro exemplo seria utilizar mais de um motor em uma mesma roda, quando um motor falhar, o controle passa a enviar comandos para o outro motor, porém devido a dificuldades construtivas, a abordagem de duplicação de motores é pouco utilizada na prática (CALDAS, 2004). Com essa técnica os erros podem ser detectados e isolados ao nível de *hardware*, assim o *software* não precisa ser alertado sobre a falha, pois está recebendo informações consistentes. Porém (VISINSKY, 1991 e FERRELL, 1993) argumentam que uma desvantagem dessa técnica é que duplicar sensores é muito caro, e torna o robô maior e mais pesado, principalmente em duplicação de atuadores. Conforme (VISINSKY, 1991 e CALDAS, 2004) o uso de mais graus de liberdade (DOF[6]) em atuadores está sendo muito utilizado, ao invés de duplicação de motores, permitindo diferentes configurações de movimentos para que o atuador possa alcançar um determinado ponto.

Outra técnica existente de tolerância a falhas é utilizar comportamentos redundantes (PAYTON *et al*, 1992 *apud* FERRELL, 1993). Suponha um robô que possui vários comportamentos programados para andar sem bater em objetos. O comportamento preferencial é andar utilizando o sensor ultra-sônico, porém se esse sensor apresentar problemas, outro comportamento para andar é ativado, por exemplo, andar utilizando um sensor infravermelho para evitar objetos. A falha é detectada se o comportamento atual possui um desempenho inferior ao esperado. Assim outro comportamento é ativado, e a estratégia é repetida até que se encontre um comportamento com um desempenho aceitável. Porém segundo (FERRELL, 1993) a desvantagem dessa técnica é que ela não é boa para falhas de

[6] DOF (*degree of freedom*), para cada direção independente que cada um dos atuadores do robô pode se mover é contado um DOF (NIEMUELLER; WIDYADHARMA, 2003).

hardware e não especifica quais são os problemas, somente os sintomas. Além disso, podem ser feitas diversas tentativas até se encontrar uma estratégia que funcione. Por exemplo, se um motor apresentar falhas, essa estratégia tentará diversos comportamentos até encontrar um que funcione bem com somente um motor. Seria mais interessante o robô ser alertado que um motor está com falhas e adaptar-se a essa situação.

2.4 TRABALHOS RELACIONADOS

Esta seção apresenta alguns trabalhos científicos utilizando o Lego Mindstorms encontrados na literatura.

(MELENDEZ; CASTILLO; SORIA, 2008) propuseram a utilização de um controle reativo utilizando lógica difusa para efetuar a programação do NXT com o objetivo de encontrar uma saída em um labirinto desconhecido. O programa desenvolvido era muito pesado para o poder de processamento do NXT, por essa razão ele optou por utilizar uma biblioteca escrita em *C++* para efetuar a comunicação por *bluetooth* com o robô, deixando toda a lógica e a decisão das ações a serem tomadas sendo executadas em um computador externo. O resultado final foi que o robô conseguiu encontrar a saída do labirinto sem bater em nenhuma parede, demonstrando a eficiência da utilização de lógica difusa para a resolução do problema do labirinto.

(KROUSTIS; CASEY, 2008) utilizaram a combinação de funções heurísticas com algoritmos adaptativos, para permitir que o NXT encontrasse uma fonte de luz em um ambiente desconhecido. Os autores montaram o robô com dois sensores de luz e dois sensores de toque, porém para a programação eles enfrentaram o mesmo problema da falta de memória do NXT. Por essa razão eles optaram por utilizar o iCommand (LEJOS, 2009), descrito na seção 2.1.1, para efetuar a comunicação via *bluetooth* com o robô.

(GEORGAS; TAYLOR, 2008), estudaram o uso de algoritmos adaptativos baseado em políticas pré-definidas e a sua aplicabilidade em robôs. Uma das experiências desenvolvidas foi a programação de um algoritmo adaptativo e a sua aplicação em um NXT. Os autores utilizaram o iCommand para se comunicar com o NXT através do *bluetooth*, deixando todo o processamento para o computador. O objetivo era simples: o robô saia de um ponto inicial, e procurava bolinhas no ambiente que estavam espalhadas, quando ele encontrava alguma, ele a recolhia e trazia para o ponto inicial de onde ele tinha saído. Os autores utilizaram um classe própria do iCommand, chamada de *Tachometer Navigation*, que armazena informações sobre os locais visitados, baseado na informação do tacômetro dos

motores, por isso ele sabia onde era o ponto inicial de partida. Porém, segundo os autores esse componente tende a falhar quando a comunicação do *bluetooth* com o computador é intensa, fazendo com que o tacômetro perca informações. A política adaptativa que eles utilizaram nessa situação é parar de utilizar a classe que contém as informações do tacômetro e começar a utilizar uma classe que utiliza o sensor ultra-sônico para localização. A classe que contém o sensor ultra-sônico não possui informações sobre o estado atual da localização. Nesse caso, primeiro o robô localiza uma parede e a segue até chegar ao ponto inicial de partida. O robô identifica que chegou ao ponto quando o índice de reflexão da parede onde ele está é diferente do resto do ambiente. Essa parede com índice de reflexão diferenciado identifica o ponto inicial de partida.

(BENITEZ; MORENO; VALLEJO, 2008), desenvolveram seu trabalho utilizando uma versão anterior do Mindstorms NXT, o RCX. Os autores verificaram que existe um problema de instabilidade na construção dos motores do RCX, dessa forma não é possível garantir que o robô se moverá em uma linha completamente reta quando estiver andando para frente, pois poderá haver diferenças (mesmo que mínimas), de rotação ou velocidade entre os dois motores. Dessa forma, os autores desenvolveram um algoritmo capaz de calibrar os motores do RCX, baseado nas informações obtidas através de um sensor de rotação. Para chegar até o algoritmo eles efetuaram vários testes, como por exemplo, quantos centímetros o robô se locomove a cada revolução de rodas, quantas revoluções são necessárias para o robô percorrer todo o ambiente, etc. Para validar o algoritmo, eles desenvolveram uma interface gráfica para demonstrar o ambiente real onde o robô era inserido, assim era possível acompanhar na tela do computador a locomoção do robô no ambiente. Os autores demonstraram que o seu algoritmo para calibrar os motores é bastante eficiente, porém depende do tamanho do caminho a ser percorrido, pois o algoritmo somente tende a diminuir os erros na localização do robô no ambiente, mas mesmo assim os sensores podem acumular erros, proporcional ao número de rotações. Como trabalhos futuros os autores sugeriram a implementação do seu algoritmo no NXT.

(PFEIFER *et al*, 2006), utilizou dois Lego Mindstorms para efetuar a programação e a simulação de robôs capazes de efetuar a coleta de lixo médico hospitalar de forma autônoma, evitando assim vários problemas que essa atividade poderia causar para a saúde humana, como doenças e contaminações. Os robôs possuem sensores que detectam quando a abertura de escoamento do lixo é aberta, assim eles se dirigem até essa abertura, coletam o lixo e o levam até uma caixa de coleta central. O autor utilizou dois Mindstorms, um deles é o mestre e o outro é escravo, ou seja, o escravo só age quando solicitado pelo mestre. O mestre é o que

está mais perto da saída do escoamento. Segundo o autor, essa relação de mestre e escravo permite um maior sincronismo nas ações. Para realizar esse trabalho o autor também utilizou a versão anterior do NXT, o RCX, assim como uma versão anterior do NXC (NXC, 2008a), especifica para o RCX.

(ARAÚJO; LIBRANTZ, 2006), construíram um robô capaz de encontrar o menor caminho para chegar a um alvo. Para isso eles utilizaram uma câmera de vídeo posicionada no teto do ambiente a ser explorado, enviando uma foto do ambiente para um computador, que transforma a imagem binária (foto digitalizada) em uma matriz. Assim ele identifica (através da foto) o robô, os obstáculos e os alvos. Para escolher o menor caminho ele utiliza funções heurísticas. A seguir, comandos são enviados ao robô para efetuar o deslocamento no ambiente. É importante ressaltar que os autores também utilizaram a versão mais antiga do NXT, a RCX para a construção do robô.

Foram encontrados também alguns artigos interessantes com a proposta de utilizar o Lego Mindstorms em disciplinas de ensino de lógica de programação (BARNES, 2002) (SCHEP; MCNULTY, 2002), e inteligência artificial (KLASSNER, 2002) (PARSONS; SKLAR, 2004), o que é uma proposta interessante, pois conforme (ROCHA, 1993 *apud* FABRI, 2008) a taxa de reprovação ou desistência em cadeiras relacionadas à lógica de programação chega a 60%, o que demonstra que são necessários métodos mais eficazes ou mais atrativos para o ensino dessas disciplinas, pois essas disciplinas são fundamentais para a formação de pessoas aptas a trabalhar nos campos relacionados à computação.

3 TRABALHOS PROPOSTOS

Este trabalho possui duas propostas. A primeira é desenvolver um sensor virtual para efetuar tratamento de falhas sensoriais e testar a sua eficiência em um robô Lego Mindstorms NXT com arquitetura *Subsumption*. Para validação desta proposta o objetivo do robô será seguir uma rota delimitada por uma linha de cor preta.

Para o robô identificar as cores, sabendo assim quando estará andando sobre a linha preta, serão utilizados dois sensores de luz (descrito na seção 2.1, Figura 6).

Conforme descrito na seção 2.3, o principio básico para desenvolver um sistema tolerante a falhas é a redundância. Sendo assim serão utilizados dois sensores de luz para permitir que de alguma forma o robô consiga identificar possíveis falhas em um dos sensores e que consiga continuar seguindo a linha, mesmo na presença de falhas. Será desenvolvido um sensor virtual que efetuará as leituras dos sensores físicos. O sensor virtual ficará responsável por identificar as falhas nos sensores físicos procurando devolver sempre valores confiáveis de leitura ao programa. Portanto, o programa não saberá da existência dos sensores físicos, nem conseguirá se comunicar diretamente com eles, a troca de informação será feita diretamente com o sensor virtual.

Um exemplo de utilização de um sistema capaz de seguir uma rota são os veículos autônomos que possuem sistemas de apoio ao motorista, ajudando-o a manter-se no centro da pista de uma rodovia (essa técnica é chamada de *Lane Keeping Assistance)*, utilizada principalmente em viagens longas, porém a abordagem utilizada para isso não é seguir uma linha e sim detectar as bordas da pista (JUNG *et al*, 2005).

Devido à limitação de sensores que são disponibilizados com o kit da Lego, não foi possível criar um ambiente mais realístico, como por exemplo, fazer com o robô se mantenha em uma estrada detectando as bordas da pista. Para a realização deste projeto foi necessário adquirir um sensor extra para programação do sensor virtual, pois o kit da Lego só disponibiliza um de cada tipo, conforme citado na seção 2.1.

A segunda proposta é efetuar o tratamento de falhas motoras, por exemplo, o robô possuir uma roda maior que a outra ou um dos motores estiver falhando. Além disso, como cada roda é controlada por um motor distinto não é possível garantir que ambos se movimentarão com exata precisão, mesmo utilizando rodas do mesmo tamanho. Nesse caso deverá ser feita uma compensação nas velocidades de cada motor a fim de obter um equilíbrio. O robô utilizado será o mesmo montado para a primeira proposta, porém o objetivo

será um pouco diferente. Para essa segunda proposta o objetivo do robô será manter-se em uma linha reta de cor preta.

Novamente por questão de limitação de sensores não será possível fazer com que o robô siga uma rota mais complexa, que seja diferente de uma linha reta. Caso houvesse disponível um sensor do tipo bússola (comprado separadamente), seria possível fazer com o robô se deslocasse em qualquer direção, e através do sensor, poderia saber se ele estaria desviando da rota principal, conseguindo assim fazer a correta compensação dos motores. Por exemplo, o robô sairia inicialmente apontando para o norte, e após estar se movendo, qualquer variação de sentido apontada pelo sensor de bússola seria indicativo que há diferença nos motores. Como só há disponível um sensor de luz, a única forma de saber se o robô está saindo da rota é fazendo com que ele siga uma linha reta, caso o robô perca a linha é sinal que um dos motores está fora de sincronia.

4 IMPLEMENTAÇÃO

Esta seção descreve toda a metodologia utilizada para a programação do robô. Na seção 4.1 serão descritas as ferramentas escolhidas para o desenvolvimento do projeto, e na seção 4.2 como foi desenvolvida a programação.

4.1 ESCOLHA DAS FERRAMENTAS

Esta seção descreve as ferramentas escolhidas para o desenvolvimento do projeto. Na seção 4.1.1 será descrita como foi feita a escolha da linguagem de programação, e na seção 4.1.2 como foi escolhido o ambiente de programação.

4.1.1 Linguagem de Programação

Conforme descrito na seção 2, o NXT possui pouca memória para armazenamento de informações, além de um poder de processamento muito mais baixo do que qualquer computador *desktop* encontrado hoje no mercado. (MELENDEZ; CASTILLO; SORIA, 2008), (KROUSTIS; CASEY, 2008) e (GEORGAS; TAYLOR, 2008) se depararam com essa dificuldade no desenvolvimento de seus trabalhos, pois utilizaram muitas técnicas de inteligência artificial, por essa razão eles tiveram que utilizar o recurso do *bluetooth* do NXT. (MELENDEZ; CASTILLO; SORIA, 2008) utilizaram uma biblioteca em *C++* para se comunicar com o NXT e (KROUSTIS; CASEY, 2008) e (GEORGAS; TAYLOR, 2008) utilizaram o iCommand, deixando assim todo o processamento do programa para o computador.

O projeto LeJOS (LEJOS, 2009), possui um boa documentação das classes que podem ser utilizadas para a programação do robô (tanto o projeto LEJOS NXJ quanto o iCommand), assim como foram encontrados alguns artigos relevantes que utilizaram essa ferramenta (KROUSTIS; CASEY, 2008) e (GEORGAS; TAYLOR, 2008). O projeto LeJOS é totalmente código aberto, dessa forma, caso necessário, é possível alterar as classes existentes, assim como estendê-las, visto que Java é totalmente orientado a objetos. Baseado nesses fatos, optou-se por utilizar o LeJOS para efetuar a programação do robô, porém haviam duas opções: utilizar o LEJOS NXJ, deixando o robô totalmente autônomo, rodando o programa diretamente no NXT, ou utilizar o iCommand e deixar toda a lógica de programação no computador, somente enviando comandos para o NXT através do *bluetooth*. Para evitar

problemas futuros, como por exemplo, que o programa ficasse muito pesado para o NXT efetuar o processamento, e ter que readaptá-lo para utilizar em outra ferramenta, o iCommand seria uma boa opção, porém precisava-se efetuar o teste de eficiência do *bluetooth*, tal como o tempo de resposta entre o envio de um comando até a sua execução. Foi encontrado um projeto de desenvolvimento de um módulo de controle remoto para o NXT, utilizando o iCommand (versão 0.7), desenvolvido por (KAHRIMANOVIC, 2008), que pode ser visualizado na Figura 12.

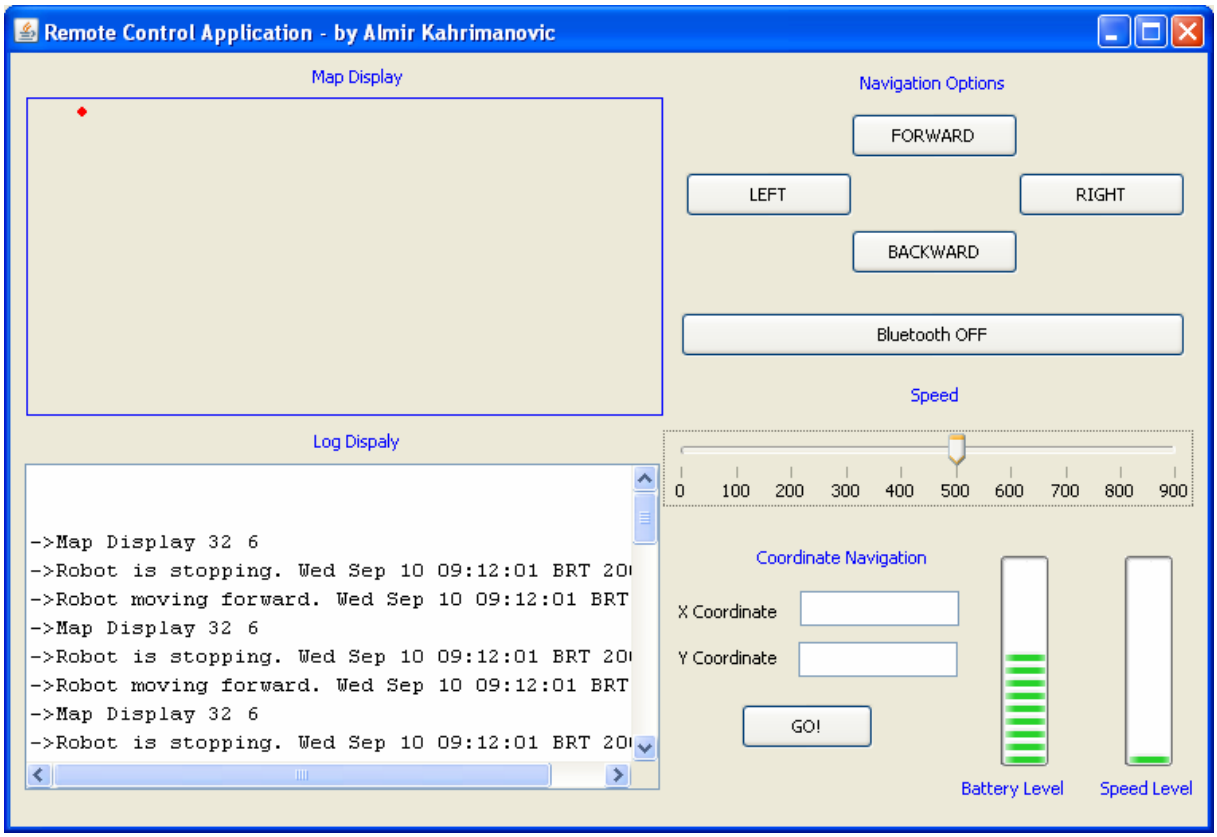

Figura 12 - Controle Remoto para o NXT (KAHRIMANOVIC, 2008).

O objetivo da aplicação é extremamente simples, somente enviando os comandos básicos para o robô, como andar para frente, para trás e virar a esquerda ou à direita.

Inicialmente, os resultados dos testes obtidos com essa ferramenta foram satisfatórios, com o NXT respondendo rapidamente aos comandos, a uma distância aceitável. Porém quando se iniciou o trabalho prático, programar o robô para andar em cima de uma linha de cor preta, verificou-se que o tempo de resposta do *bluetooth* não era tão satisfatório para a tarefa proposta, pois o tempo de resposta era fundamental, ou seja, quando o robô perdesse a linha, outro comportamento deveria ser acionado. No caso prático, assim que o programa identificava que o robô tinha perdido a linha um comando de parar era enviado ao mesmo, e até que o robô recebesse o comando e o processasse, ele já tinha se afastado da linha em

aproximadamente cinco centímetros. Pesquisando novamente na bibliografia, Brian Bagnall (BAGNALL, 2007) co-autor do *firmware* LeJOS, explica que inicialmente o NXT parece ter pouca memória, principalmente com os computadores atuais na casa dos Gigabytes de memória, porém robôs não necessitam de muita memória, pois não precisam rodar programas gráficos nem efetuar processamento muito pesado (com exceção de programas que utilizam Inteligência Artificial e mapeamento de ambientes). Como exemplo, ele cita que os robôs da NASA *Spirit* e *Opportunity* possuíam apenas 256 Kilobytes de memória *flash*! Bagnall (2007, p.5) ainda afirma: "Se foi memória suficiente para a NASA, você pode confiar que há memória suficiente para os seus projetos". Pelas razões descritas, o LeJOS (versão 0.8) foi escolhido para o desenvolvimento desse projeto e conseqüentemente a linguagem de programação será Java.

4.1.2 Ambiente de Programação

Como a linguagem de programação escolhida foi Java, existem diversas opções de ambientes integrados para desenvolvimento (IDE[7]) para escolha, algumas proprietárias como IntelliJ IDEA, Borland JBuilder e JCreator e outras *freeware* como NetBeans, BlueJ e Oracle JDeveloper. Existe uma IDE código aberto denominada Eclipse (ECLIPSE, 2008), que é bastante interessante não só por ser de código aberto, mas também por prover um ambiente completo para desenvolvimento, além de permitir adicionar novas funcionalidades através de *plug-ins*[8] que podem ser desenvolvidos ou adquiridos na Internet. O projeto LeJOS, por exemplo, possui um *plug-in* para o Eclipse que permite sobrescrever o *firmware* do NXT com o do LeJOS diretamente pela IDE com apenas um clique. O Eclipse pode ser visualizado na Figura 13.

Segundo (CHEN; MARX, 2005) a IDE Eclipse torna o desenvolvimento em Java mais eficiente e produtivo, pois possui uma série de recursos como auxilio a sintaxe da linguagem, depurador de código, sugestões para auto-completar o código que se está digitando e vários assistentes durante o desenvolvimento de novos projetos ou classes. Além disso, ela possui uma série de recursos que podem ser utilizados por programadores profissionais, como, por exemplo, controle de versões e gerenciamento de projetos.

Conforme (DEUGO, 2008) o Eclipse é a IDE preferida de mais da metade dos programadores Java, o que demonstra uma alta aceitação da mesma pelos desenvolvedores.

---

[7] IDE: *Integrated Development Environment* ou em português ambiente integrado para desenvolvimento.
[8] Programa que serve para adicionar funcionalidades a outros programas maiores.

Por ser uma ferramenta sem custo para aquisição e bastante completa, tendo inclusive um *plug-in* pronto para o projeto LeJOS que permite que os programas sejam enviados rapidamente para o NXT, além de ser utilizada inclusive por programadores profissionais, a IDE Eclipse foi escolhida para o desenvolvimento deste projeto.

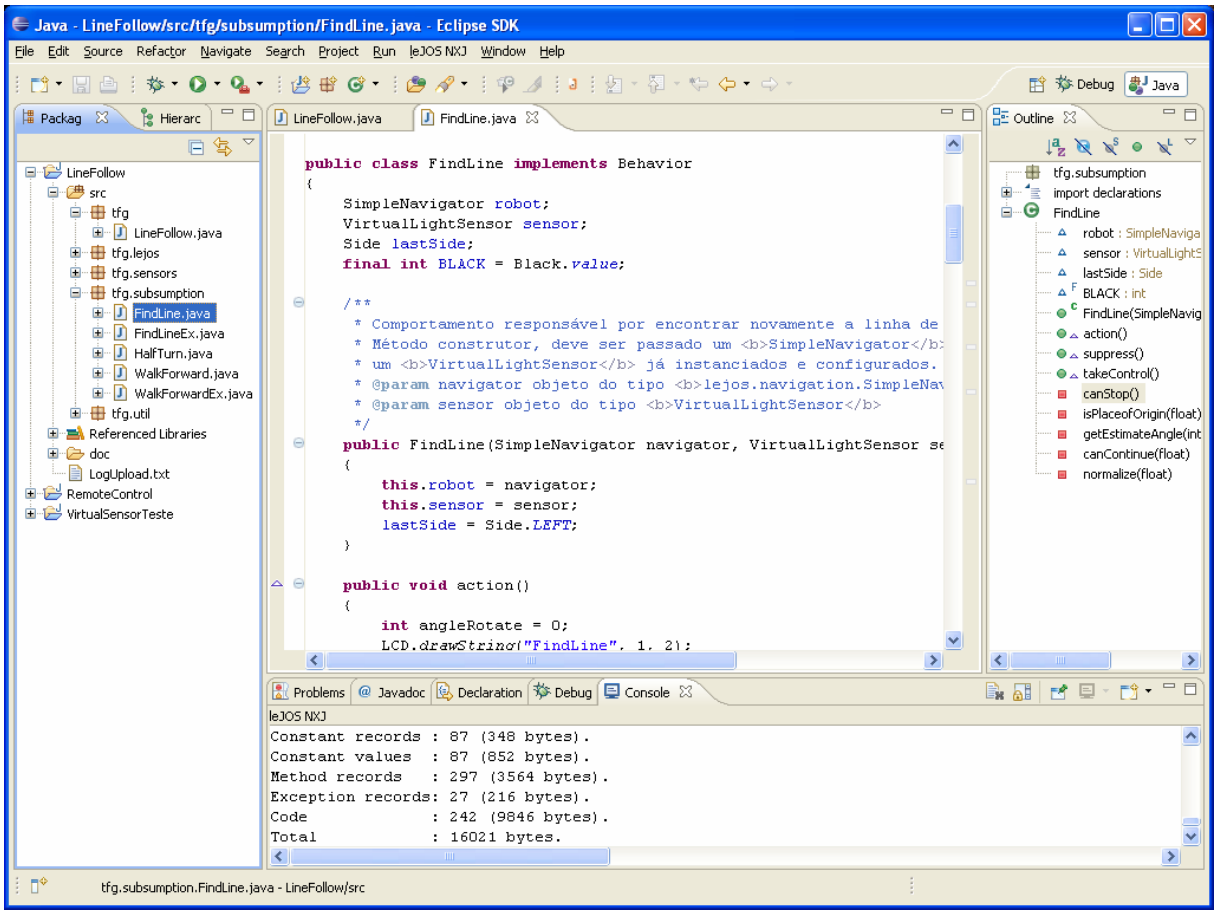

Figura 13 – Ambiente Integrado de Desenvolvimento Eclipse.

4.2 PROGRAMAÇÃO

Após definida a linguagem de programação, iniciou-se a leitura da documentação do projeto LeJOS, e constatou-se que a mesma já possui suporte para a programação de robôs baseados em comportamentos, através da arquitetura *Subsumption* (Seção 2.2.1).

Para dar suporte a arquitetura *Subsumption*, o projeto LeJOS definiu uma interface chamada *Behavior* (comportamento), que possui os métodos abstratos da Tabela 1.

Tabela 1 - Métodos da Interface *Behavior*

RETORNO	NOME	DESCRIÇÃO
void	Action	Ação que o robô deverá executar quando o comportamento for acionado.
void	Suppress	O código que será executado quando outro comportamento for acionado, ou seja, quando o comportamento atual tiver que parar.
boolean	takeControl	Método que deverá possuir um teste para verificar se o comportamento deve ser acionado.

Por ser uma interface, nenhum método está implementado. Para utilizar a interface basta criar um classe que a implemente, assim o programador será obrigado a definir o código dos três métodos da Tabela 1. A Figura 14 exemplifica como fica a estrutura básica da classe.

```
import lejos.subsumption.Behavior;

public class WalkForward implements Behavior {

    public void action() {
        // Código
    }

    public void suppress() {
        // Código
    }

    public boolean takeControl() {
        // Código
        return false;
    }
}
```

Figura 14 - Classe que implementa a interface *Behavior*.

O pacote *Subsumption* do projeto LeJOS ainda conta com uma classe responsável por arbitrar a troca dos comportamentos, a classe *Arbitrator*. Para isso o programador deve criar um vetor de *Behaviors*, sendo que comportamentos de menor prioridade devem ser colocados nas posições iniciais do vetor. Após isso, o vetor deve ser passado para o arbitrador, que por sua vez fica executando um laço de repetição varrendo o vetor, executando o método *takeControl* de cada comportamento, e caso retorne verdadeiro, o método *suppress* do comportamento atual é executado, e o novo comportamento é ativado, através do método

action. Caso dois comportamentos precisem ser ativados ao mesmo tempo, o controle é dado ao comportamento de maior prioridade (definido pela posição no vetor de comportamentos).

Como o código do projeto LeJOS é aberto, foi feita uma modificação na classe *Arbitrator* pois o laço de repetição que varria os comportamentos era infinito. Com a modificação efetuada, basta clicar no botão "sair" do *brick* para que a execução do programa seja interrompida.

4.2.1 Sensor Virtual

Conforme descrito na seção 2.3, uma das técnicas utilizadas para tolerância a falhas em robótica é o uso de sensores virtuais, que são sensores abstratos que embutem a detecção e o tratamento de falhas.

O sensor virtual desenvolvido para este projeto é uma classe Java que gerencia dois sensores de luz físicos. As requisições de leituras são feitas ao sensor virtual, que por sua vez lê os valores dos sensores físicos. No caso de não haver falhas, ou seja, os dois sensores estarem funcionando corretamente é retornado a média das leituras, caso contrário deve ser identificado qual dos sensores está com falha. O fluxo de execução do sensor virtual pode ser visualizado na Figura 15.

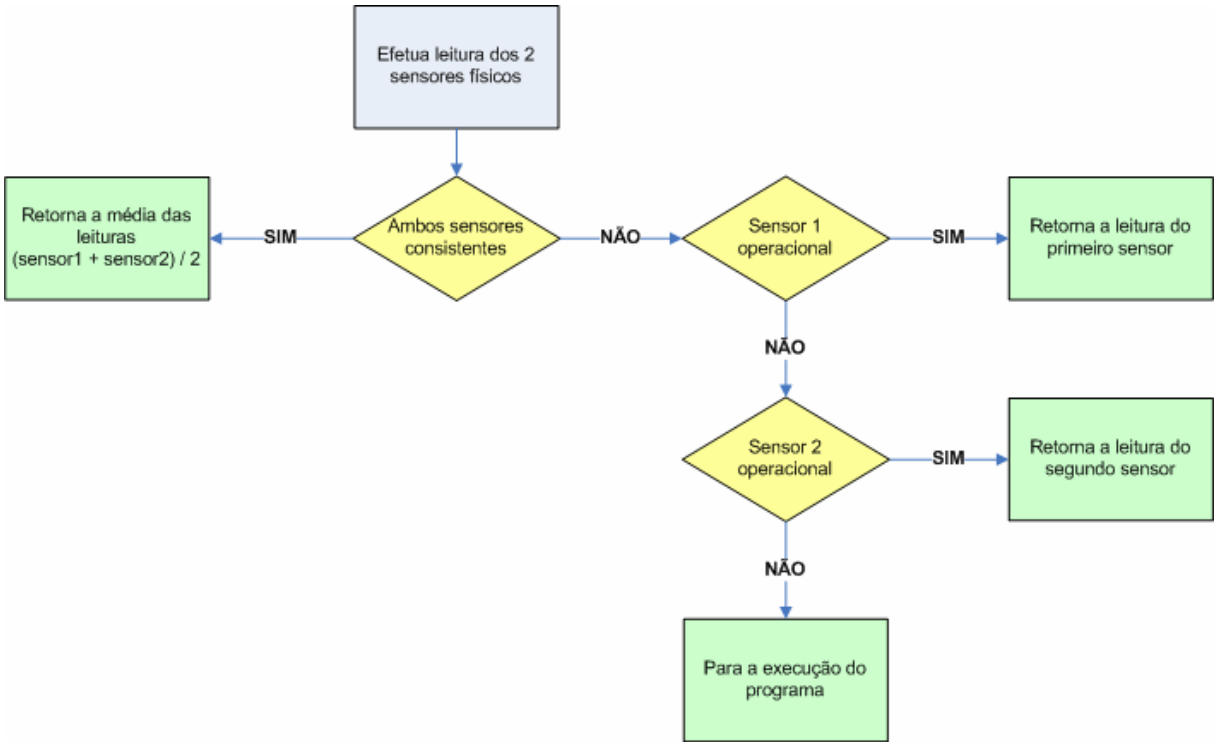

Figura 15 - Fluxograma de execução do sensor virtual

A verificação da consistência dos sensores visualizada no primeiro ponto de decisão do fluxograma tem dois aspectos: intervalo de leitura e repetição. O Intervalo atual de leitura deve estar entre -30 e 130 (dados obtidos através de amostragem; depois de calibrados, teoricamente a leitura dos sensores deve ficar entre 0 (preto) e 100 (branco), porém dá-se uma margem de tolerância, pois não é possível garantir que o sensor foi calibrado utilizando o ponto mais escuro e o mais claro do trajeto) . O segundo aspecto é a repetição, ou seja, se possui menos de 100 leituras repetidas (as 100 últimas). Pelos testes efetuados, sempre ocorrerem variações nas leituras, a menos que o robô esteja parado. Assim mais do que 100 leituras repetidas no mesmo sensor são um indicio forte de que há problema. O valor 100 parece ser alto, porém conforme pesquisa no fórum de suporte do LeJOS (LEJOSFORUM, 2009), Andy Shaw, pseudônimo *gloomyandy*, um dos responsáveis pelas atualizações do *firmware* do LeJOS, explica que os valores dos sensores são lidos a cada 2 milisegundos e armazenados em *cache* no *firmware*. Assim novos valores estão disponíveis a cada 2 milisegundos. O sensor virtual é acionado ininterruptamente pelos comportamentos, portanto 100 leituras representam 20 milisegundos, ou 0,02 segundos. É possível concluir através desses dados que 100 leituras é um valor aceitável para verificação de repetição.

É importante ressaltar que todos esses passos são executados para cada solicitação de leitura ao sensor virtual. Nesse caso quando um sensor é marcado como inválido, pois sua leitura está fora da faixa, ou possui mais de 100 leituras repetidas, na próxima solicitação de leitura ele pode vir a ser marcado como válido, pois parou de repetir os valores ou seu valor está novamente dentro da faixa. Este comportamento está correto, pois conforme descrito na seção 3, através do trabalho de Visinsky (1991), pode haver discrepâncias nos valores lidos por ruídos no ambiente (muita luz em determinado local, relevo no chão, etc.), ou seja, uma falha transitória.

```
/** Instancia os sensores de luz físicos */
LightSensor luz1 = new LightSensor(SensorPort.S1);
LightSensor luz2 = new LightSensor(SensorPort.S2);
/** chama um método que efetua a calibragem dos sensores  */
calibraSensorLuz(luz1, luz2);
/** Cria um novo sensor virtual, passando os
 * sensores físicos calibrados como parâmetro  */
VirtualLightSensor sensor = new VirtualLightSensor(luz1, luz2);
/** sensor virtual pronto para ser utilizado como
 * exemplificado abaixo:  */
int valorLido = sensor.getLightPercent();
```

Figura 16 - Utilizando a classe do sensor virtual.

Além disso, pelos testes efetuados, caso os sensores estejam realmente com falha ou defeito, como por exemplo, desligados ou queimados, além dos valores continuarem sempre repetidos, eles estarão fora da faixa permitida, o que irá garantir o correto tratamento do mesmo, não havendo risco de serem marcados como válidos incorretamente. A Figura 16 demonstra como instanciar e utilizar a classe do sensor virtual (*VirtualLightSensor*).

4.2.2 Sensoriamento – Andar sobre a linha (*LineFollow*)

Para validação do projeto de programar o robô que seja capaz de seguir uma linha, foi desenvolvido um programa denominado *LineFollow*, e três comportamentos que implementam a interface *Behavior*: *WalkForward* (andar para frente), *HalfTurn* (meia volta) e *FindLine* (encontrar linha).

Na Figura 17 é possível visualizar como ficou a hierarquia de comportamentos conforme arquitetura *Subsumption*.

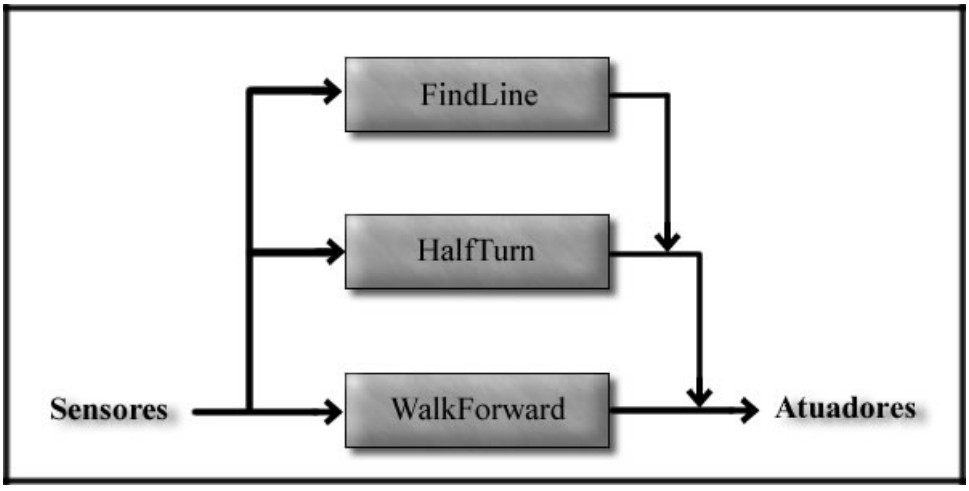

Figura 17 - Hierarquia de comportamentos.

O comportamento *WalkForward* é o que possui menor prioridade, e é o mais simples dos três. Sua função é somente andar para frente, enquanto estiver sobre a linha preta, ou seja, enquanto o sensor virtual devolva valores abaixo de 20 unidades. Conforme explicado na seção 4.2.1, depois de calibrados a leitura dos sensores deve ficar entre 0 (preto) e 100 (branco), porém não é possível garantir que o sensor foi calibrado no ponto mais escuro do trajeto. Portanto uma margem de tolerância é necessária. Pelo testes efetuados, quando o sensor está em cima da cor preta, o valor máximo retornado foi em torno de 20 unidades.

O comportamento *HalfTurn* não possui um papel fundamental para o funcionamento do programa, é um comportamento auxiliar que serve para inverter a direção do robô, ou seja, dar uma volta de 180 graus em torno do próprio eixo. Foi utilizado para fins de testes, para verificar se o robô conseguia encontrar a linha andando nos dois sentidos, ou seja, mesmo que fosse invertida a direção atual. Ele é acionado quando se aperta os botões "esquerda" ou "direita" do *brick*. Caso o comportamento *FindLine* esteja em execução, o *HalfTurn* não poderá ser acionado, pois possui menor prioridade.

Finalmente o comportamento *FindLine* é responsável por encontrar novamente a linha preta. O robô considera que não está andando em cima da linha se o valor retornado pelo sensor virtual for maior que 20, neste caso o comportamento é ativado. É o comportamento mais importante, pois nada adianta andar para frente ou dar uma volta de 180 graus se tiver perdido a linha que é o principal objetivo do robô, consequentemente possui a prioridade mais alta.

Para encontrar novamente a linha, a lógica utilizada é girar o robô para esquerda e para direita, aumentando gradualmente o ângulo de rotação, até que a linha seja encontrada. Valores de ângulo positivos fazem com o que o robô gire para a esquerda e valores negativos para a direita. Inicia-se a rotação com 10 graus. A primeira vez que o comportamento é executado, a tentativa inicial de achar a linha será para a esquerda. A regra de qual lado girar nas próximas execuções do comportamento será descrita no próximo parágrafo. Caso o robô não encontre a linha, são somadas 10 unidades ao valor absoluto do ângulo atual e multiplicado por -1, ou seja, na segunda interação ele irá girar 20 graus para a direita, após 30 graus para a esquerda e assim sucessivamente até o limite de 40 graus.

Outro fator importante desenvolvido é que depois de encontrada a linha, o robô armazena para qual lado ele estava girando quando a encontrou. Se a linha foi encontrada girando para a direita, na próxima execução do comportamento o ângulo inicial será -10, ao invés de 10. Se fosse fixado sempre começar para a esquerda a tentativa inicial de encontrar a linha, o robô perderia desempenho caso estivesse fazendo uma curva acentuada para a direita, ou seja, ele sempre faria uma rotação desnecessária para a esquerda para depois girar para direita e finalmente encontrar a linha. Além disso, se for uma curva acentuada a tendência é de o robô voltar a perder a linha na mesma curva.

A maioria dos métodos existentes no projeto LeJOS que fazem com que o robô se movimente possuem um parâmetro opcional denominado *immediateReturn*. Caso esse parâmetro seja verdadeiro (*true*), o método retorna imediatamente, em outras palavras, o comando para movimentar-se é enviado ao robô e o programa não fica parado esperando o

movimento ser concluído, ou seja, permite que outros métodos executem concorrentemente durante o movimento. Assim, a cada rotação para encontrar a linha o parâmetro *immediateReturn* é passado como verdadeiro, fazendo com que enquanto o robô está efetuando a rotação, outro método é executado concorrentemente para verificar se durante a rotação ele já encontrou a linha. Nesse caso a rotação é interrompida e o robô para, dando a possibilidade do comportamento *WalkForward* ou *HalfTurn* assumirem o controle.

Se a linha não for encontrada em até 40 graus de rotação, significa que a curva é um ângulo muito fechado. Neste caso a estratégia adotada é girar até 180 graus para um dos lados, a partir do ponto inicial de onde a linha foi perdida. Se a linha for encontrada antes de completar o giro de 180 graus é sinal que a curva foi completada e o robô continua o deslocamento. Se a linha for detectada apenas ao final dos 180 graus, o robô está sobre o eixo original, mas no sentido oposto. Neste caso é necessário verificar se não existe linha para outro lado.

Um problema que surgiu durante essa fase do projeto é que quando era enviado um comando para o robô girar os 180 graus, ao final da rotação o ângulo que o robô informava ter girado ficava em torno de 174 graus. Isso dificultou os testes para verificar se o robô estava sobre o eixo original. Para utilizar as classes de navegação do projeto LeJOS, é necessário informar dois parâmetros: o tamanho da roda (*wheelDiameter*) e a distância entre as rodas (*trackWidth*). O valor do primeiro parâmetro está informado na própria roda que vem com o kit: 5,6 cm. Para medir o valor do segundo parâmetro, a documentação do LeJOS informa que deve ser medida a distância do centro de uma roda até a outra, nesse caso o valor obtido foi de 11,3 cm. Como o valor do ângulo esperado não fechava com o obtido, foi feita uma análise no código fonte das classes de navegação e descobriu-se que a fórmula utilizada para cálculo do ângulo é a seguinte:

$$\hat{A}ngulo = \frac{(RotacoesMotor1 - RotacoesMotor2) \times TamanhoRoda}{2 \times DistânciaEntreRodas}$$

O tamanho da roda estava correto, pois estava impresso nas rodas que acompanham o kit. O valor das rotações de cada motor também estava correto, pois é uma informação obtida diretamente do motor. Assim o único valor suspeito era a distância entre as rodas que foi medido manualmente. Como todos os valores da fórmula eram conhecidos, assim como o valor esperado de ângulo, isolou-se a variável *DistânciaEntreRodas* e obtivemos o valor

correto para fechar o cálculo do ângulo: 10,9 cm, ou seja, 0,4 cm de diferença do valor medido inicialmente.

É possível visualizar na Figura 18 que os sensores ficaram localizados na frente das rodas. A montagem dessa forma foi necessária pela técnica utilizada para encontrar a linha, que é girar o robô para os lados. Numa montagem inicial, os sensores ficaram localizados muito atrás, sendo que o sensor mais a frente havia ficado posicionado na mesma linha das rodas. Nos testes efetuados o robô tinha dificuldades em encontrar a linha quando somente o sensor da frente estava ativo, e não encontrava a linha quando utilizava o outro sensor. Isto se deve ao fato da área abrangida pelos sensores ser muito pequena no momento das rotações, assim quanto mais para frente os sensores ficarem, mais fácil será de encontrar a linha.

Figura 18 - NXT montado para o projeto *LineFollow*

4.2.3 Tolerância motora (*MotorCompensate*)

Para validação da tolerância motora, foi criado um programa denominado *MotorCompensate*, no qual o objetivo do robô é se manter em cima de uma linha de cor preta. Os comportamentos utilizados foram os mesmos do projeto *LineFollow*, porém, como presume-se que o robô está se deslocando em uma linha reta, o comportamento *FindLine* nunca deveria ser chamado. Assim, caso esse comportamento seja executado é sinal que algum dos motores está com diferença de velocidade ou uma roda com diferença de tamanho. Quando o robô encontra a linha, outro método é executado para tentar ajustar a velocidade

individual de cada motor. Como existe esse procedimento adicional para ajustar a velocidade dos motores, o comportamento *FindLine* foi substituído por outro denominado *FindLineEx*.

O projeto LeJOS possui um método dentro da classe que controla os motores, que tenta sincronizar a velocidade dos motores automaticamente, baseado no sensor de rotação interno de cada motor. Como o objetivo desse projeto é simular que o robô está andando com uma roda maior que a outra, a sincronização automática de velocidade inviabilizaria a realização desse projeto. Apesar de a sincronização ser padrão, ela pode ser facilmente desativada através da chamada do método *regulateSpeed()*, passando *false* como parâmetro.

A velocidade máxima que pode ser utilizada em cada motor, caso as pilhas estejam com carga máxima é de 900. Um método foi desenvolvido para que possa ser selecionada a velocidade inicial desejada antes de chamar o programa, porém para testes sempre foi utilizada uma velocidade inicial de 200.

O comportamento *FindLineEx* é uma extensão do comportamento *FindLine* utilizado no projeto *LineFollow*. Portanto, a técnica utilizada para encontrar a linha é exatamente a mesma descrita no outro projeto: rotacionar o robô alternadamente para a esquerda e para a direita, aumentando gradualmente o ângulo de rotação, até que a linha seja encontrada. O robô sempre inicia o deslocamento em cima da linha preta, e caso ele perca a linha para um dos lados, o comportamento *FindLineEx* é executado, e rotaciona até encontrar a linha. Após a linha ser encontrada, a velocidade dos motores é ajustada (compensada).

A velocidade de cada motor é compensada dentro do comportamento *FindLineEx* da seguinte forma: se o robô encontrou a linha quando estava girando para a esquerda a velocidade do motor da direita que deve ser compensada e vice-versa. Enquanto for o mesmo motor que está sendo compensado na chamada do método, a velocidade é aumentada de 20 em 20 unidades. Por exemplo, inicialmente a velocidade do motor da direta e da esquerda é de 200. O robô encontra a linha girando para a esquerda, isso significa que o robô perdeu a linha por sair para a direita. Esse fato foi causado por um movimento maior da roda esquerda. Assim, a velocidade do motor da direita é ajustada para 220, a fim de tentar equilibrar o movimento maior que a roda esquerda está exercendo. Se na próxima execução do comportamento, o robô encontra a linha girando para a esquerda, novamente a velocidade do motor da direita deve ser aumentada, passando para 240, pois o motor da esquerda ainda está executando um movimento maior. Porém, se dessa vez a linha foi encontrada girando para a direita, o motor da esquerda não pode ser compensado em 20 unidades, pois estaria voltando à situação inicial com os dois motores na mesma velocidade. Outra questão importante de ressaltar é que se na primeira vez que ele perdeu a linha foi para o lado direito, aumentou-se a

velocidade do motor direito e agora ele perdeu para a esquerda, é sinal que o aumento de velocidade foi muito alto, e agora o outro lado deve ser equilibrado, porém com um valor abaixo do último. O procedimento adotado nesse caso é aumentar a velocidade do motor com 75% do valor usado no aumento anterior, feito no motor oposto. Ou seja, no exemplo citado, o motor da esquerda seria compensado em 15 unidades e não 20. A partir desse momento cada incremento é diminuído 75%. Na terceira vez que precisar de compensação, o incremento será de 11 (11.25, porém o LeJOS só trabalha com valores inteiros para velocidade) e assim por diante. Esse procedimento é repetido até que o equilíbrio de velocidade dos motores seja alcançado.

A técnica desenvolvida demonstrou ser eficiente em falhas permanentes, ou seja, o robô está com diferença no tamanho das rodas, ou com diferença nos motores (principalmente por questões mecânicas) e permanecerá assim durante toda a execução da tarefa. No caso de falhas transitórias, por exemplo, algum objeto se prendeu a roda fazendo com que ela ande mais a cada giro, o robô tentará compensar a velocidade da outra roda. Esse comportamento é correto. Porém se em algum momento o objeto se desprender da roda, o robô não identificará que a roda voltou ao normal e que deve voltar à velocidade inicial. Nessa situação o robô tentará compensar a velocidade da outra roda que estava normal.

5 RESULTADOS

Para testar a eficiência do sensor virtual, foram desenhados dois circuitos a mão livre para o robô percorrer, conforme pode ser visualizado na Figura 19.

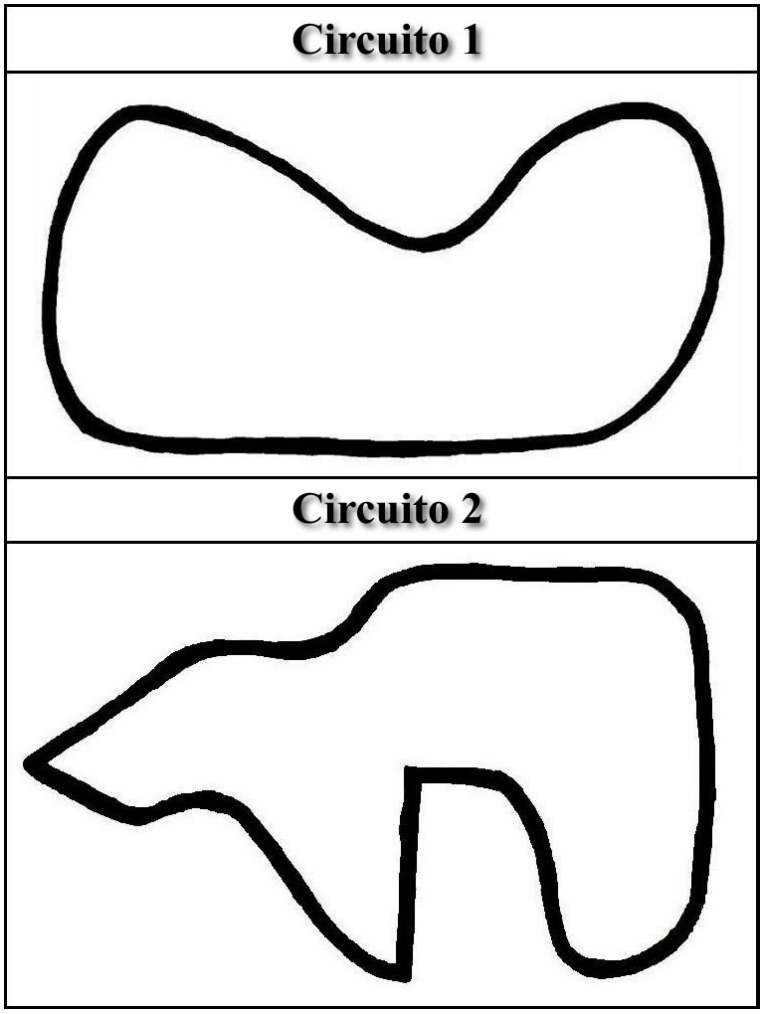

Figura 19 - Circuitos desenhados para o robô percorrer.

Para testar a validade da proposta, os sensores foram desconectados em momentos aleatórios durante o trajeto (somente um sensor era desconectado por vez, deixando sempre um operacional), para verificar se o sensor virtual conseguia identificar a falha e adaptar-se a situação, sem interromper o correto funcionamento do programa, ou seja, continuar encontrando a linha como se os dois sensores estivessem operacionais. Os resultados foram muito satisfatórios com o sensor virtual identificando instantaneamente a falha do sensor e parando de utilizá-lo. Quando o sensor era novamente conectado ao robô, o sensor virtual identificava que ele estava novamente operacional e voltava a usar a média de leitura. Utilizar

à média das leituras aumentou a confiabilidade do programa, pois se um dos sensores estivesse em cima do preto a probabilidade do sensor virtual devolver um valor até 20 unidades era grande, o que estava correto, pois somente um dos sensores havia perdido a linha.

Dois vídeos que demonstram o robô deste projeto percorrendo os circuitos da Figura 19 e identificando falhas podem ser assistidos no site de compartilhamento de vídeos *YouTube* nos endereços <http://www.youtube.com/watch?v=m99oKAnIVAs> e <http://www.youtube.com/watch?v=Mix-79NQeO0>.

Para o segundo projeto, de tolerância motora, inicialmente havia sido construído um tabuleiro em linha reta, pois mesmo as rodas sendo do mesmo tamanho, o robô sempre pendia para um lado. Uma das explicações para isso é que mesmo as rodas sendo do mesmo tamanho, os motores são distintos, por questões mecânicas e de construção não é possível garantir que os dois irão andar perfeitamente sincronizados. Porém houve certa dificuldade em validar o projeto nesse tabuleiro, principalmente pelo fato que não ser possível garantir que quando o robô encontrasse novamente a linha, ele estaria perfeitamente alinhado nela.

A forma encontrada para validação foi simular que o robô estava andando com uma roda maior que a outra. Como as rodas eram do mesmo tamanho, a solução foi desenhar um tabuleiro no formato de um círculo. Caso o robô esteja com uma roda maior que a outra, e a velocidade dos motores seja igual, a tendência é que o robô ande em círculo, pois uma das rodas estará fazendo um movimento maior. Por esse motivo foi escolhido um tabuleiro no formato de círculo para simular a diferença nas rodas.

Com esse circuito o resultado foi bastante satisfatório. Após o robô ter perdido a linha em pequenos trechos, ele compensa a velocidade dos motores, até que em certo ponto ele consegue percorrer cerca de metade do círculo sem qualquer interrupção.

6 CONCLUSÕES

O sensor virtual demonstrou ser uma alternativa eficiente para tratamento de falhas em sensores. Além disso, todo o comportamento e o tratamento de falhas podem ser encapsulados nele. Caso haja alguma falha a aplicação não será notificada e o sensor virtual se adaptará a nova situação, não interrompendo a execução do programa. É importante ressaltar que apesar do sensor virtual se adaptar as falhas é possível que ocorra uma pequena perda de desempenho. No caso especifico do robô implementado, a perda do sensor mais a frente implicou em maior dificuldade para o robô encontrar a linha. Isto se deve ao fato do outro sensor estar localizado mais próximo ao eixo das rodas, sendo sua área de cobertura menor durante os giros para encontrar a linha.

Apesar de poucos testes terem sido realizados no projeto de tolerância motora, a abordagem utilizada de compensar individualmente os motores demonstrou ser bastante válida para falhas permanentes, principalmente quando se utilizou um circuito em círculo para simular uma roda maior que a outra. Nesse circuito o robô conseguiu percorrer cerca de metade do circuito sem perder a linha.

Apesar da limitação de sensores e de *hardware* do tijolo programável, o robô da Lego demonstrou ser uma boa alternativa de baixo custo para testar programas em robôs reais, além de ser fácil de manipular e programar, tendo a possibilidade de ser programado em diferentes linguagens. Um ponto negativo observado, que é uma limitação da linguagem escolhida e não do NXT, é a inexistência de um depurador de código. Muito tempo foi despendido para a construção dos programas, pois quando se precisou depurar o código para encontrar erros, a alternativa utilizada era mandar o robô parar até que algum botão fosse pressionado e mostrar o valor de variáveis na tela de LCD.

Outro ponto negativo é a de não existir nenhum simulador para o NXT. Isso dificultou o andamento do projeto, pois somente era possível testar os programas diretamente no robô. Como o robô só estava disponível na instituição, o acesso e tempo para testar foram limitados.

É importante ressaltar que esse foi o primeiro trabalho cientifico utilizando o Lego Mindstorms na UNIFRA, dessa forma muito tempo precisou ser despendido para pesquisa e testes até que pudesse ser iniciado realmente o trabalho proposto. Como por exemplo, é possível citar a descoberta de que o iCommand não seria uma boa alternativa para o projeto conforme discutido na seção 4.1.1, o parâmetro extra (*immediateReturn*) existente nos métodos que fazem com o que o robô se movimente, discutido na seção 4.2.2, etc. Assim,

espera-se que esse trabalho também sirva como guia para próximos trabalhos científicos que utilizarem o Lego Mindstorms.

Como trabalhos futuros sugerem-se testar a eficiência do uso de sensores virtuais com sensores físicos heterogêneos, como por exemplo, um sensor virtual para detecção de objetos que consiste em um sensor ultra-sônico e um sensor de infravermelho (comprado separadamente, pois não acompanha o kit). Quando o sensor ultra-sônico falhar o sensor virtual utilizará o infravermelho para calcular a distância de objetos e vice-versa.

Outra proposta é utilizar um sensor do tipo bússola (comprado separadamente), para permitir a programação de objetivos mais complexos para o robô, sendo possível inclusive testar a tolerância motora em outros ambientes.

Por fim, é interessante explorar a tecnologia *bluetooth* do NXT aliada com a programação em Java, construindo programas que se comuniquem ou rodem em dispositivos móveis como celulares e *smarthphones*.

REFÊRENCIAS BIBLIOGRÁFICAS

ASIMOV. **Isaac Asimov Home Page**. Disponível em: <http://www.asimovonline.com>. Acesso em: 15/10/2008.

ARAÚJO, Sidnei Alves de; LIBRANTZ, André Felipe Henriques. **Navegação autônoma de robôs**. Exacta. São Paulo. v.4. n. especial. p. 81 - 83. 2006.

BAGNALL, Brian. **Maximum Lego NXT: Building Robots with Java Brains**. Variant Press, 2007.

BARNES, David J. **Teaching introductory Java through LEGO MINDSTORMS models**. In: Proceedings of the 33rd SIGCSE technical symposium on Computer science education. p 147 - 151. Cincinnati, Kentucky, USA. 2002.

BENITEZ, Antonio; MORENO, Cristian Josué; VALLEJO, Daniel. **Localization Control for LEGO Robot's Navigation**. In: 18th International Conference on Electronics, Communications and Computers. CONIELECOMP 2008. p. 187 - 192. Puebla, Mexico. 2008.

BROOKS, Rodney. **A robust layered control system for a mobile robot**. In: IEEE Journal of Robotics and Automation. v. 2, n.1, p.14 – 23. 1986.

_____. **Elephants Don't Play Chess**. In: Robotics and Autonomous Systems. n.6, p. 3 - 15. 1990.

_____. **Intelligence Without Reason**. In: Proceedings of 12th International Joint Conference on Artificial Intelligence. p. 569 – 595. Sydney, Australia. 1991.

CALDAS, Wilton Speziali. **Tolerância a falhas adaptativa para robôs móveis com arquitetura híbrida**. 2004. 263p. Tese (Doutorado em Ciência da Computação) - Curso de Pós-Graduação em Ciência da Computação. Universidade Federal de Minas Gerais, Belo Horizonte, 2004.

CHEN, Zhixiong; MARX, Delia. **Experiences with Eclipse IDE in programming courses**. Journal of Computing Sciences in Colleges. Vol. 21, No. 2. p. 104 - 112. 2005.

COELHO, Henrique; ALPOIM, Samuel; LOPES, Miguel. **Programação de robô LEGO MindStorms com o Microsoft Robotic Studio**: Trabalho prático de Sistemas Inteligentes. Universidade do Minho – PT. 2007. Disponível em: <http://henrike.coelho.googlepages.com/siTpra.pdf>. Acesso em: 09/08/2008.

DEUGO, Dwight. **Using eclipse in the classroom**. In: Proceedings of the 13th annual conference on Innovation and technology in computer science education (ITiCSE 2008). p. 322 - 322. Madrid, Spain. 2008.

DHEIN, Guilherme. **Integrando deliberação e reatividade em uma arquitetura de agentes híbrida homogênea**. 2000. 98p. Dissertação (Mestrado em Ciência da Computação) - Programa de Pós-Graduação em Ciência Computação, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, 2000.

ECLIPSE. **Eclipse.org home**. 2008. Disponível em: <http://www.eclipse.org/>. Acesso em: 01/11/2008.

EDSINGER, Aaron Ladd. **Robot Manipulation in Human Environments**. 2007. 228p. Ph.D. Thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science. 2007.

FABRI, José Augusto. **Engenharia de Software**: **Como motivar o aluno nas disciplinas introdutórias da área de programação?**. 2008. Disponível em: <http://engenhariasoftware.wordpress.com/2008/04/15/como-motivar-o-aluno-nas-disciplinas-introdutorias-da-area-de-programacao/>. Acesso em: 09/08/2008.

FERRELL, Cynthia. **Robust agent control of an autonomous robot with many sensors and actuators**. 1993. 165p. Thesis (Master of Science). Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science. 1993.

_____. **Failure recognition and fault tolerance of an autonomous robot**. Adaptive Behavior, Vol. 2, No. 4. p. 375 - 398. 1994.

GATES, Bill. Um Robô em cada casa. **Scientific American Brasil**. Edição Especial n. 25. p 6 – 13. 2007.

GEORGAS, John C.; TAYLOR, Richard N. **Policy-Based Self-Adaptive Architectures: A Feasibility Study in the Robotics Domain**. In: Proceedings of the 2008 ACM/IEEE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008). p. 105 - 112. Leipzig, Germany. 2008.

JUNG, Cláudio Rosito *et al***. Computação Embarcada: Projeto e Implementação de Veículos Autônomos Inteligentes**. In: XXIV Jornadas de Atualização em Informática (JAI 2005). Anais do XXV Congresso da Sociedade Brasileira de Computação, 2005. cap.4. p.1358 - 1406.

KAHRIMANOVIC, Almir. **Remote control software application**: Final Year Project Report. School of Computing. Dublin Institute of Technology. 2007/2008. Disponível em: <http://www.tiim.info/the_nxt_project/web/index.php>. Acesso em: 30/08/2008.

KIM, Seung Han; JEON, Jae Wook. **Programming LEGO mindstorms NXT with visual programming.** In: International Conference on Control, Automation and Systems 2007. p. 2468 - 2472. COEX, Seoul, Korea. 2007.

_____. **Using visual programming kit and LEGO Mindstorms: An introduction to embedded system**. In: IEEE International Conference on Industrial Technology (ICIT 2008). p. 1 - 6. Chengdu, China. 2008.

KLASSNER, Frank. **A case study of LEGO Mindstorms'™ suitability for artificial intelligence and robotics courses at the college level**. In: Proceedings of the 33rd SIGCSE technical symposium on Computer science education. p. 8 - 12. Cincinnati, Kentucky, USA. 2002.

KROUSTIS, Constantinos A.; CASEY, Matthew C. **Combining Heuristics and Q-learning in an Adaptive Light Seeking Robot**. Computing Sciences Report CS-08-01. University of Surrey. Guildford, UK. 2008.

LABVIEW. **LEGO® MINDSTORMS® NXT - Powered by NI LabVIEW - Academic - National Instruments**. 2008. Disponível em: <http://www.ni.com/mindstorms/>. Acesso em: 07/08/2008.

LEJOS. **LeJOS, Java for Lego Mindstorms**. 2009. Disponível em: <http://lejos.sourceforge.net/>. Acesso em: 23/05/2009.

LEJOSFORUM. **reading a light sensor**. In: leJOS.org Forum Index. 2009. Disponível em: <http://lejos.sourceforge.net/forum/viewtopic.php?t=1546>. Acesso em: 14/06/2009.

MAES, Pattie. **The agent network architecture (ANA)**. ACM SIGART Bulletin, Vol. 2, No. 4. p. 115 - 120. 1991.

MELENDEZ, A.; CASTILLO, O.; SORIA, J. **Reactive control of a mobile robot in a distributed environment using fuzzy logic**. In: Fuzzy Information Processing Society. NAFIPS 2008. p. 1 - 5. New York City, NY, USA. 2008.

MINDSTORMS. **Lego Mindstorms NXT**. 2008. Disponível em: <http://mindstorms.lego.com/>. Acesso em: 07/08/2008.

MSROBOTICS. **Conheça o Microsoft Robotics Studio**. 2008. Disponível em: <http://msdn.microsoft.com/pt-br/library/cc580631.aspx> . Acesso em: 30/08/2008.

NIEMUELLER, Tim; WIDYADHARMA, Sumedha. **Artificial Intelligence - An Introduction to Robotics**. 2003. Disponível em: <http://session.niemueller.de/uni/roboticsintro/AI-Robotics.pdf> Acesso em: 15/11/2008.

NXC. **NBC/NXC - NeXT Byte Codes and Not eXactly C**. 2008a. Disponível em: <http://bricxcc.sourceforge.net/nbc/> . Acesso em: 09/08/2008.

_____. **NXC Programmer's Guide**. 2008b. Disponível em: <http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf >. Acesso em: 09/08/2008.

PARSONS, Simon; SKLAR, Elizabeth. **Teaching AI using LEGO Mindstorms**. In: Proceedings of the AAAI Spring Symposium on Accessible Hands-on Artificial Intelligence and Robotics Education. Stanford University, California, USA. 2004.

PFEIFER, Erick *et al*. **Coleta de Lixo Médico Utilizando Protótipos de LEGO**. In: III Encontro Nacional de Robótica Inteligente (EnRI). Anais do XXVI Congresso da SBC 2006. Campo Grande, MS, Brasil. 2006.

ROBOTC. **ROBOTC.net.** 2008. Disponível em: http://www.robotc.net/ . Acesso em: 04/10/2008.

RUSSEL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. Prentice Hall, 1995.

SCHEP, Madeleine; MCNULTY, Nieves. **Use of Lego mindstorm kits in introductory programming classes: a tutorial**. In: Journal of Computing Sciences in Colleges. Vol. 18. p. 323 - 327. Consortium for Computing Sciences in Colleges, USA. 2002.

SILVA JÚNIOR, Edson Prestes e. **Navegação Exploratória baseada em Problemas de Valores de Contorno**. 2003. 109p. Tese (Doutorado em Computação) - Curso de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

SOMANI, Arun K.; VAIDYA, Nitin H. **Understanding Fault Tolerance and Reliability**. Computer. Vol. 30, No. 4. p. 45 - 50. IEEE Computer Society. 1997.

TAYLOR, Russel H.; STOIANOVICI, Dan. **Medical robotics in computer-integrated surgery**. In: IEEE Transactions on Robotics and Automation. Vol. 19. Issue 5. p. 765 - 781. 2003.

TECHNIC. **LEGO.com TECHNIC Start**. 2008. Disponível em: http://technic.lego.com . Acesso em: 30/08/2008.

VALDEJÃO, Renata de Gaspari. **Robôs são parceiros do médico na sala de cirurgia**. FTC Digital. 23/06/2008. Disponível em: <http://blog.ftc.br/ftcdigital/?p=13>. Acesso em: 28/09/2008.

VERMA, Vandi *et al*. **Real-time fault diagnosis [robot fault diagnosis]**. In: IEEE Robotics & Automation Magazine. Vol. 11. Issue 2. p. 56 - 66. 2004.

VISINSKY, Monica Lynn. **Fault Detection and Fault Tolerance Methods for Robotics**. 1991. 103p. Thesis (Master of Science). Rice University. Houston, Texas. 1991.

_____.; CAVALLARO, Joseph R.; WALKER, Ian D. **A Dynamic Fault Tolerance Framework for Remote Robots**. In: IEEE Transactions on Robotics and Automation. Vol. 11. Issue 4. p. 477 - 490. 1995.

WOOLDRIDGE, Michael. **Intelligent Agents**. In: Multiagent Systems: Modern Approach to Distributed Artificial Intelligence. Edited by Gerhard Weiss. The MIT Press, 1999. cap.1. p.1 - 51.