Plataforma de Software para Robôs Móveis Autônomos

Silas F. R. Alves[1], João M. Rosário[2], Humberto Ferasoli Filho[3]

[1] Dep. de Engenharia Elétrica, Escola de Engenharia de São Carlos, Universidade de São Paulo
Avenida Trabalhador São-carlense, 400, 13566-590 - São Carlos – SP
salves@sc.usp.br, insilva@sc.usp.br
[2] Dep. de Projeto Mecânico, Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Caixa Postal 6122, 13083-970 Campinas, São Paulo, Brasil
rosario@fem.unicamp.br
[3] Departamento de Computação, Faculdade de Ciências, Universidade Estadual Paulista
Caixa Postal 473, 17033-360 Bauru, São Paulo, Brasil
ferasoli@fc.unesp.br

Abstract. Diferentes técnicas de navegação para robôs móveis foram desenvolvidas nas últimas décadas. Entretanto, experimentar tais técnicas não é uma tarefa trivial, pois usualmente não é possível reusar o software de controle desenvolvido devido a incompatibilidades dos sistemas. Este artigo propõe uma plataforma de software que fornece os meios para criar módulos de software reusáveis através da padronização de interfaces de software que representam os vários módulos de um robô móvel.

Keywords: Navegação de Robôs Móveis, Plataforma de Software, Engenharia de Software

Nível: Mestrado, defendido em 16/12/2011; Habilitado para o CTDR 2012.

1 Introdução

A navegação é um aspecto importante da robótica móvel, pois confere aos robôs móveis as habilidades básicas de interação com o ambiente e os obstáculos e agentes nele situados. Diferentes técnicas de navegação para robôs móveis foram desenvolvidas nas últimas décadas, entretanto, experimentar estas técnicas não é uma tarefa corriqueira, pois pode envolver adequações mecânicas, de hardware ou de software. Adequar a estrutura mecânica ou o hardware de um robô móvel é uma tarefa complexa, cara e demorada. Por outro lado, a adequação de software é menos trabalhosa graças às ferramentas e linguagens de programação, muito embora não exista uma solução de desenvolvimento ótima.

Ainda que as arquiteturas de software dos robôs comerciais ofereçam componentes de software que abstraem a aquisição de dados sensoriais e o controle de atuadores, há pouco suporte nativo para técnicas de navegação. Comumente, as técnicas de navegação são implementadas pelo usuário, pois é ele quem conhece todos os requisitos da aplicação e as restrições do robô móvel. Contudo, se por um lado a obrigatoriedade de

escrever métodos de navegação e colaboração resulte na escolha do método que melhor se adapte a aplicação do robô móvel, por outro lado esta obrigação dificulta o desenvolvimento da aplicação do robô. Neste caso, o usuário deve escrever tanto os algoritmos referentes à aplicação quanto os referentes à navegação do robô. Além disso, no que pauta a pesquisa nesta área, o usuário deve implementar um ou mais métodos tradicionais de navegação a fim de compará-los a novos métodos em desenvolvimento.

Algumas abordagens encontradas na literatura incluem o Robot Operating System (ROS) [1], as ferramentas de software Player/Stage [2], e o projeto OROCOS [3]. Estes projetos fornecem interfaces para sensores e atuadores comuns, bem como para algumas técnicas de navegação no caso do ROS. Estas interfaces são independentes de hardware e, portanto, o programa do usuário não precisa ser modificado quando o hardware é trocado se o novo hardware possuir as interfaces utilizadas. Outra característica importante é a modularidade destas abordagens. Uma interface modular fornece flexibilidade, permitindo ao usuário facilmente substituir módulos similares sem modificar muito código fonte.

Neste trabalho, foi desenvolvida uma plataforma de software modular para robôs móveis que tira proveito de técnicas de programação para propiciar flexibilidade ao software do usuário. Ao contrário do ROS e Player/Stage que desenvolvem interfaces para diferentes módulos dos robôs móveis, este trabalho é voltado para o desenvolvimento específico de interfaces para módulos de navegação, como técnicas de localização ou representação de mapas. Por esta razão, a modelagem de sensores e atuadores não foi considerada neste estágio do projeto.

O objetivo principal deste trabalho é o desenvolvimento de uma plataforma de software que facilite a criação de aplicativos de controle para robôs móveis através da padronização das interfaces de software para a Navegação entre Robôs Móveis. Com esta plataforma, é possível portar o software para diferentes robôs com os mesmos componentes de software, o que reduz o tempo de desenvolvimento do aplicativo ao incentivar o reuso de software. Além disso, as técnicas de navegação fornecidas pela plataforma amenizam o esforço em desenvolver o software de controle para robôs móveis colaborativos. Isso é possível porque a plataforma permite que o usuário concentre seus esforços na solução dos problemas pertinentes a aplicação do robô, uma vez que as técnicas de navegação e colaboração são fornecidas pela plataforma.

2 Arquitetura Proposta

De forma geral, os computadores constituem hoje o módulo responsável pela tomada de decisão do robô e pela arbitração dos demais módulos. Muito embora seja possível criar robôs que não tenham um computador embarcado, esta abordagem resulta em um robô pouco flexível que foge da proposta da arquitetura deste trabalho.

O computador que controla o robô móvel pode ser embarcado ou remoto. Quando embarcado no robô, o computador é capaz de acessar o hardware do robô através de um barramento local. No modo remoto, o robô móvel possui embarcado um sistema simples – normalmente um supervisor – responsável por acionar o hardware do robô

quando solicitado pelo computador remoto. A conexão entre o computador e o supervisor se dá através de um link de comunicação que pode ser com fio ou sem fio. Ambos os arranjos são mostrados pela Fig. 1.

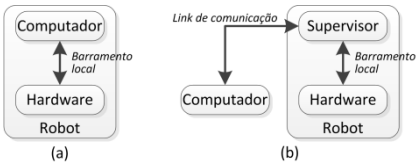

Fig. 1. Computador (a) embarcado e (b) remoto.

O computador necessita ser programado, o que envolve questões de desenvolvimento de software como a adoção da arquitetura de software que define não apenas o modelo de processamento adotado, mas também a facilidade de implementação, manutenção e reuso de código. Como o objetivo principal deste trabalho é desenvolver uma plataforma de software que facilite a criação de aplicativos para robôs móveis, o planejamento da plataforma contemplou três características para o software, baseadas nos Fatores de Qualidade de McCall [4]: flexibilidade, reusabilidade e portabilidade. Para alcançar estas características, a plataforma desenvolvida – batizada de TAO – faz uso de interfaces padronizadas de software que implementam diferentes módulos da navegação e colaboração entre robôs móveis. Os módulos contemplados atualmente pela plataforma são: localização, representação de caminho, e controle.

Esta seção apresenta os métodos adotados e o desenvolvimento do trabalho.

2.1 Arquitetura de Software

Na literatura, é possível encontrar diferentes abordagens para a composição de arquiteturas de software para robôs móveis. De forma geral, elas podem ser classificadas como bibliotecas [2, 3], sistemas operacionais (SO) [5], middleware [6] ou baseadas em protocolos de redes de computadores [7]. Para o desenvolvimento da plataforma TAO, optou-se pela arquitetura baseada em bibliotecas, por não requerem comunicação entre processos (IPC), o que reduz a complexidade do algoritmo e evita o custo de tempo para troca de mensagens. Também foi escolhida a linguagem C++, pois é robusta, aberta e com suporte a Orientação a Objetos (OO). O uso de OO, aliado ao conceito de polimorfismo [8], confere maior flexibilidade ao promover o reuso de código, característica muito utilizada no desenvolvimento desta arquitetura.

2.2 Sistema Operacional

Outra escolha tecnológica diz respeito ao SO utilizado. Neste trabalho foi contemplada apenas uma família de SO, o GNU/Linux, com o uso de bibliotecas POSIX (*Portable Operating System Interface*, ou Interface Portável ente Sistemas Operacionais) para melhorar sua portabilidade entre sistemas UNIX. O GNU/Linux designa uma família de SOs de código fonte aberto, com várias distribuições voltadas a dife-

rentes aplicações, e que contempla diferentes plataformas de hardware [9]. Em decorrência desta escolha, a TAO pode ser executada em qualquer plataforma robótica que forneça suporte a uma distribuição do GNU/Linux, seja de forma remota ou embarcada. Para o desenvolvimento deste trabalho, foi adotada a distribuição Ubuntu em sua versão 10.04 de 32 bits, por ser uma distribuição popular e madura.

2.3 Interfaces de Software

Neste trabalho, cada módulo de um sistema robótico móvel foi mapeado por uma classe que serve como interface padronizada para tais módulos. Esta decisão foi tomada com o intuito de padronizar as interfaces para facilitar e incentivar o reuso de código. Uma interface consiste em uma classe que contém definições, ou "rascunhos", dos métodos que devem ser desenvolvidos nas classes descendentes que implementam tal interface. Em outras palavras, todas as classes descendentes de uma interface possuem métodos com o mesmo nome, mesmos parâmetros e mesma funcionalidade, porém os códigos destes métodos são diferentes para cada classe. Junto ao polimorfismo, estas interfaces permitem que suas classes descendentes sejam trocadas ou reusadas por outras classes sem a necessidade de reescrever o código fonte.

Para exemplificar como as interfaces foram utilizadas, suponha duas interfaces: *Tração* e *Localização*. A interface *Tração*, mostrada pela Fig. 2 (a), contém os métodos necessários para acionar um sistema de tração qualquer, seja ele com rodas ou não, através das velocidades descritas pelo robô. Já a interface *Localização*, mostrada pela Fig. 2 (b), fornece os métodos necessários para verificar qual a localização estimada por um sistema de localização qualquer. Considere, também, quatro classes hipotéticas: *Diferencial* e *Holonômico*, mostradas respectivamente pela Fig. 2 (c) e Fig. 2 (d), que representam dois sistemas de tração diferentes e que implementam a interface *Tração*; *Odometria* e *Visão Global*, mostradas respectivamente pela Fig. 2 (e) e Fig. 2 (f), que representam dois sistemas de localização diferentes e que implementam a interface *Localização*.

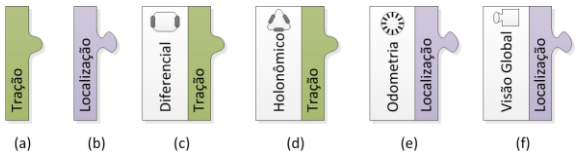

Fig. 2. Interfaces para sistema de (a) *Tração* e (b) *Localização*, duas classes de tração (c) Diferencial e (d) *Holomômico*, e duas classes de localização por (e) *Odometria* e (f) *Visão Global*.

Por implementarem a mesma interface *Tração*, tanto a classe *Diferencial* quanto a classe *Holonômico* fornecem os métodos padrão de *Tração*, ao passo que *Odometria* e *Visão Global* fornecem os métodos padrão de *Localização*. Logo, qualquer objeto de *Diferencial* pode ser substituído por um objeto de *Holonômico*, da mesma forma que um objeto de *Odometria* pode ser substituído por um objeto de *Visão Global*. Contudo, um objeto de *Diferencial* não pode, por exemplo, substituir um objeto de *Odometria*, pois implementam diferentes tipos de interface.

Com as interfaces *Tração* e *Localização*, é possível escrever códigos que fazem uso dos métodos por elas fornecidos. Por exemplo, suponha a classe *Controlador*, mostrada pela Fig. 3 (a), responsável por controlar a locomoção de um robô móvel. A classe *Controlador* utiliza um sistema de localização – fornecido pela interface *Localização* – para recuperar a posição atual do robô móvel, e um sistema de tração – fornecido pela interface *Tração* – para locomover o robô. Por usar as interfaces padrão, a classe *Controlador* pode utilizar as classes *Diferencial*, *Holonômico*, *Odometria* e *Visão Global* sem necessitar a adequação de seu código. Para utilizá-las, a classe Controlador requer apenas que as classes desejadas sejam referenciadas nas interfaces corretas. Por exemplo, a Fig. 3 (b) mostra a classe *Controlador* utilizando as classes *Diferencial* e *Odometria* referenciadas em suas respectivas interfaces, enquanto a Fig. 3 (c) mostra a mesma classe Controlador utilizando as classes *Holonômico* e *Visão Global*. Para que houvesse a modificação do sistema de tração e localização, não foi necessário modificar o código fonte da classe *Controlador*. Para modifica-lo, foi necessário apenas trocar as classes referenciadas pelas interfaces.

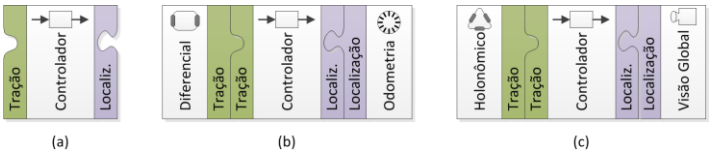

Fig. 3. (a) Classe *Controlador*, com dois exemplos de uso: (b) utilizando as classes *Diferencial* e *Odometria*, e (c) com as classes *Holonômico* e *Visão Global*.

2.4 Modelo Cinemático do Robô

Os robôs móveis adotados neste projeto utilizam um sistema de tração diferencial. Considerando C como o centro de massa do robô, r como o raio da roda, R como a distância entre as rodas, $\dot{\varphi}_d$ e $\dot{\varphi}_e$ como as velocidades angulares das rodas direita e esquerda, v e ω como as velocidades linear e angular do robô, e θ como a orientação do robô, tem-se o modelo cinemático conforme a Eq. (1). Este é um modelo já difundido na literatura e é discutido detalhadamente por Siegwart e Nourbakhsh [10].

$$\begin{bmatrix} \dot{\varphi}_d \\ \dot{\varphi}_e \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & \frac{R}{r} \\ \frac{1}{r} & -\frac{R}{r} \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{1}$$

2.5 Estabilizadores de Posição e Trajetória

O estabilizador de posição é um controlador capaz de levar de uma posição qualquer da área de trabalho até uma meta. A lei de controle se dá conforme descrita por Alves [11], com a adição de uma regra que impede o robô de mover-se quando o erro angular θ_e é maior que um valor pré-estabelecido k_1, e é mostrada pelas Eq. (2-3).

$$\omega_c = k_2 \cdot \theta_e \tag{2}$$

$$v_c = \begin{cases} v_{\max}, & d_e > d_a, \ |\theta_e| < k_1 \\ v_{\max} \cdot \frac{d_e}{d_a}, & d_e \leq d_a, \ |\theta_e| < k_1 \\ 0, & |\theta_e| \geq k_1 \end{cases} \quad (3)$$

O Estabilizador de Trajetória adotado implementa o controlador apresentado por Siciliano e Khatib [12]. O objetivo deste controlador é estabilizar o erro de postura do robô em zero, dada a trajetória de referência e a posição do robô, ambas definidas no tempo. Conforme discute Siciliano e Khatib [12], a lei de controle é dada por:

$$w_1 = -k_1|v_r|(z_1 + z_2 z_3), \ k_1 > 0 \quad (4)$$

$$w_2 = -k_2 v_r z_2 - k_3|v_r|z_3, \quad k_2, k_3 > 0 \quad (5)$$

2.6 Visão Global

A Visão Global é uma técnica de localização global que utiliza as imagens adquiridas de uma câmera localizada paralela ao solo para identificar a posição e orientação do robô dadas por (x, y, θ). Esta técnica de localização é mais precisa que a odometria e pode fornecer informação sobre outros objetos presentes na área de trabalho. Para facilitar o reconhecimento dos robôs, cada robô possui um marcador fidedigno único com duas cores diferentes. Este sistema é semelhante ao utilizado por competições de futebol de robôs.

2.7 Mapeamento e Planejamento de Caminho

Os mapas tratam do problema da representação do mundo e dos objetos e seres que nele existem – inclusive o robô móvel. Em geral, os mapas podem ser classificados em: *geométricos*, quando o ambiente é representado através das geometrias dos obstáculos nele presentes[12]; ou *topológicos*, quando não se utiliza uma medida precisa, baseando-se principalmente em navegação por marcos cujas relações geográficas são representadas tipicamente por grafos [13]. Complementarmente, os mapas geométricos podem ser discretizados em mapas de grade de ocupação. Este trabalho adotou os mapas de grade de ocupação por apresentarem pouca complexidade computacional e serem de fácil modelagem.

O planejamento de caminho no mapa em grade de ocupação pode ser realizado através do método do espaço das velocidades ou pelo método do espaço geométrico [14]. Outra forma de abordar este problema é através do uso de algoritmos de busca, como os algoritmos de Djkstra, A^* e D^* [15]. Neste trabalho, foi utilizado o algoritmo A^* devido sua simplicidade de implementação.

3 Módulos Desenvolvidos

As classes desta arquitetura foram divididas em um pacote chamado *Navigation2D*, com as classes que implementam a navegação no plano.

3.1 Navigation2D

O pacote *Navigation2D* é composto por duas estruturas básicas e três subpacotes. As estruturas básicas são: *Pose*, que grava as informações sobre a pose do robô com a tripla (x, y, θ); e *Speed*, que grava informações de velocidade em termos de $(\dot{x}, \dot{y}, \dot{\theta})$. Estas estruturas são utilizadas pelas demais classes do pacote *Navigation2D*, presentes nos subpacotes, para representar as poses e velocidades do robô. Já os quatro subpacotes são: *Localization* (Localização), que fornece as interfaces para implementar as técnicas de localização; *Locomotion* (Locomoção), com as interfaces necessárias para implementar a locomoção do robô; e *Path* (Caminho), que oferece suporte a construção de caminhos.

O pacote *Localization* (Localização) contém a classe de interface *LocalizationTechnique* (Técnica de Localização) que serve como classe ancestral para todas as técnicas de localização implementadas dentro desta arquitetura. Portanto, ela oferece métodos padronizados para ler a posição e velocidade do robô. Há também um método para configurar a posição do robô, que é utilizado por sistemas de localização inercial ou para calibrar o offset de sistemas de localização absolutos.

O pacote *Path* (Caminho) é responsável por implementar as classes e interfaces referentes à geração de caminhos ou trajetórias. Para isso, foram desenvolvidas duas estruturas para representar caminhos, uma em termos de posição, *PathPose*, e outra em termos de velocidade, *PathSpeed*. A classe *PathFinder* é a interface que dita os métodos que uma função de busca de melhor caminho deve oferecer. Para exemplificar o uso desta interface e também fornecer um algoritmo de busca para a arquitetura, foi criada a classe *AStar*, descendente de *PathFinder*, que implementa o algoritmo A^*. É importante notar que a classe *PathFinder* – e, consequentemente, a classe *AStar* – recebem como parâmetro um mapa do tipo *GridMap* e retornam um caminho descrito pela classe *PathPose* ou *PathSpeed*.

Finalmente, o pacote *Locomotion* (Locomoção) fornece uma interface para o sistema de tração de um robô móvel, uma interface para sistemas de controle de locomoção, e duas classes para controle de locomoção. A interface para o sistema de tração do robô, *DriveSystem*, fornece os métodos necessários para acionar os motores do robô através das velocidades $(\dot{x}, \dot{y}, \dot{\theta})$. Todo robô que utiliza esta arquitetura deve fornecer uma classe descendente de *DriveSystem* para o acionamento dos motores do robô, que implemente, também, os métodos definidos pela classe ancestral.

Já a interface *DriveControl* serve como classe ancestral para todas as classes que implementam um controle de locomoção e, portanto, a classe *PositionStabilizer*, que implementa o controlador da seção 3.5, e a classe *TrajectoryStabilizer*, que implementa o controlador da seção 3.6, dela descende.

O diagrama de classes mostrado pela Fig. 4 destaca as classes anteriormente apresentadas e seus relacionamentos.

3.2 Pacotes Desenvolvidos para os Robôs Móveis

Para que os robôs 14-bis e Roburguer fossem capazes de usufruir da arquitetura TAO, foi necessário criar classes de acionamento para os robôs descendentes da clas-

se *DriveSystem*. Também foi criada uma classe descendente de *LocalizationTechnique* para o sistema de Visão Global descrito na seção. Como todos os robôs da BER utilizam sistema de tração diferencial, foi crida a classe *DiffDrive*, descendente de *DriveSystem*, para cada um dos robôs. Para o sistema de Visão Global, foi criada a classe *GlobalVision*, descendente de *LocalizationTechnique*, que faz a interface com o software criado para o AEDROMO. Esta classe oferece as posições de dois robôs e de um objeto presente na área de trabalho.

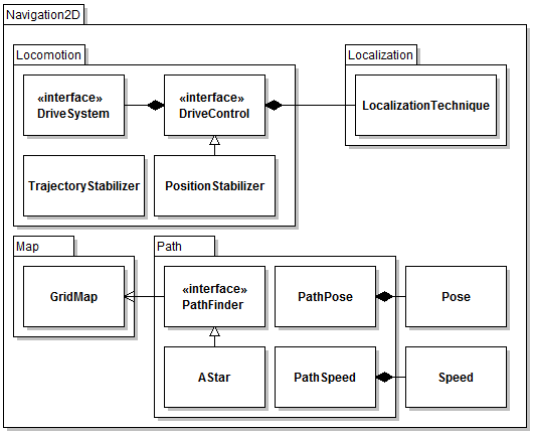

Fig. 4. Diagrama de Classes simplificado da plataforma.

Também foi criada a classe *Simulator*, que implementa um simulador. Esta classe descende tanto de *DriveSystem* quanto de *LocalizationTechnique*. Para o software do usuário, a classe *Simulator* funciona da mesma forma que uma classe escrita para um robô real. Desta forma, o código utilizado na simulação é o mesmo utilizado nos testes reais, não requerendo, portanto, que sejam criadas duas versões diferentes do software em ambientes de desenvolvimento diferentes. Esta flexibilidade acelera o desenvolvimento do software e facilita sua manutenção.

4 Experimentos e Resultados

Para avaliar e demonstrar o funcionamento da TAO, foram realizados alguns testes com o estabilizador de posição implementado pela classe *PositionStabilizer*. Para tanto, foram utilizados dois robôs móveis que serão descritos a seguir.

4.1 Robôs móveis utilizados

Para realizar os experimentos, foram considerados dois robôs de tração diferencial: 14-bis e Roburguer, mostrados pela Fig. 5. Ambos foram desenvolvidos para atender aplicações variadas, como educação, entretenimento e reabilitação. O desenvolvimento e avaliação destes robôs é discutido por Ferasoli et al. [16].

Fig. 5. Foto dos robôs 14-bis (esquerda) e Roburguer (direita) [16].

Estes robôs utilizam um sistema de locomoção baseado no modelo cinemático apresentado na seção 4.4 e fornecem dois sensores reflexivos para detecção de linha. O robô 14-bis também possui uma caneta, que é utilizada para desenhar nas superfícies, enquanto o robô Roburguer fornece um mecanismo que o permite mover pequenos objetos.

4.2 Discussão dos Experimentos e Resultados

Para testar o controlador discreto implementado pela classe *PositionStabilizer*, foi realizado um teste onde o robô deve se dirigir ao centro da área de trabalho, dado pelas coordenadas $(0.4\,,0.3)$, partindo de um ponto qualquer. Na Fig. 6 (a), apresentam-se cinco repetições deste teste, onde a posição de partida do robô varia em cada uma das iterações. Conforme pode ser observado na mesma figura, o robô foi capaz de alcançar o centra da área de trabalho em todas as iterações do teste.

O erro de posição, dado pela distância entre a posição atual do robô e o centro da área de trabalho, é apresentado pela Fig. 6 (b). Nela, pode-se observar que o controlador apresenta o mesmo comportamento em todas as iterações do teste. Assim que o controlador entra em ação, o erro se mantém constante até o momento em que o robô se alinha com o ponto central da área de trabalho. Uma vez alinhado, o controlador aciona o robô em sua velocidade máxima, causando a redução constante do erro até que este erro seja igual a $0{,}05$ m. Quando o erro se torna menor que $0{,}05$ m, o controlador passa a desacelerar o robô, o que diminui a taxa de redução do erro. No fim de cada iteração, o erro torna-se inferior ao critério de parada do controlador, que equivale a $0{,}001$ m. Portanto, pode-se concluir que o controlador discreto implementado pela classe *PositionStabilizer* obteve resultados satisfatórios nos experimentos.

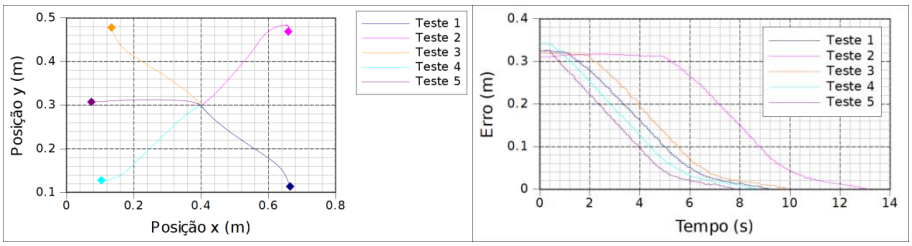

Fig. 6. Gráficos com (a) Trajetória descrita pelo robô nos cinco testes e (b) erro de posição em cada um dos cinco testes.

O controlador de trajetória *TrajectoryStabilizer* também foi experimentado. Dada a trajetória de referência oval e sabendo que a posição inicial do robô de referência é $(0{,}24\ ,\ 0{,}15\ ,\ 0)^{\mathrm{T}}$, enquanto a posição inicial do robô móvel real é dada por $(0{,}244\ ,\ 0{,}127\ ,\ -0{,}02)^{\mathrm{T}}$, obteve-se a trajetória mostrada pela Fig. 7. É possível observar que o robô é capaz de estabilizar-se sobre a trajetória, apesar de apresentar pequenos desvios de posição decorrentes da dificuldade do robô em manter as velocidades instantâneas desejadas e do atraso do sistema de Visão Global, que é de cerca de 200 ms

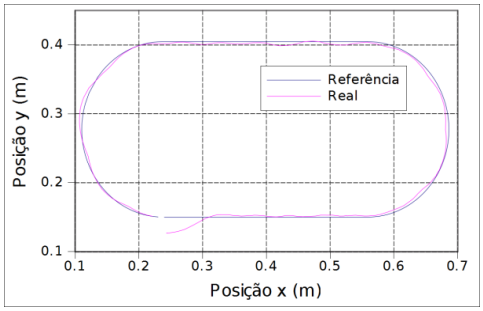

Fig. 7. Comparação entre a trajetória de referência e a trajetória descrita pelo robô móvel.

Também foram avaliadas as classes *GridMap* para criação de mapas de grade de ocupação e *AStar* para busca de melhor caminho. Para isso, foi elaborado um mapa de grade de ocupação a ser representado pela classe *GridMap*. Como a área de trabalho do robô mede 0,8 m por 0,6 m e o corpo do robô tem um diâmetro de 0,1 m, o mapa foi divido em grades quadradas de 0,1 m de lado. Estabeleceram-se neste mapa as grades inicial e final para o robô, que são, respectivamente, $(1, 1)$ e $(6{,}1)$. Tendo o mapa e as grades inicial e final como parâmetros, foi utilizada a classe *AStar* para gerar o caminho. Tanto o mapa quanto o caminho gerado é mostrado pela Fig. 8 (a).

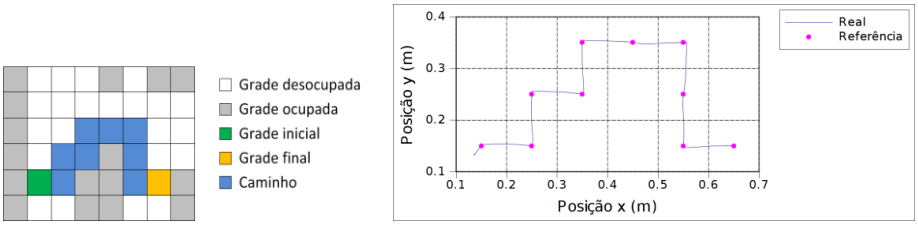

Fig. 8. (a) Mapa do experimento e caminho gerado, e (b) caminho descrito pelo robô móvel.

Finalmente, com o caminho gerado, foi possível utilizar a classe *PositionStabilizer* para controlar o robô móvel dentro da papelaria hipotética, conforme mostra a Fig. 8 (b). Para completar o caminho da Fig. 8 (b), o robô levou 60,5 s. Neste teste, o robô seguiu satisfatoriamente o caminho gerado pela classe *AStar*. O caminho evitou colisões com os obstáculos estáticos presentes no ambiente e completou a execução num tempo aceitável. A Fig. 9 mostra ambos os robôs durante os experimentos, conforme vistos pela câmera de visão global.

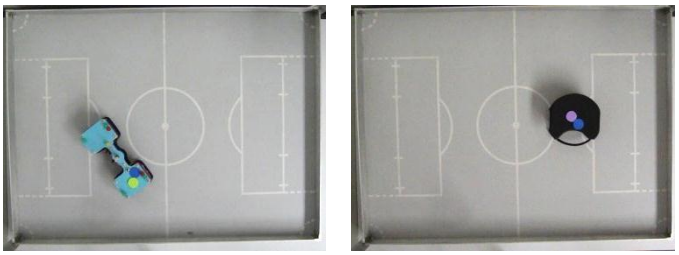

Fig. 9. Robôs 14-bis (esquerda) e Roburguer (direita) durante os experimentos.

5 Conclusão

A plataforma desenvolvida, TAO, facilita o desenvolvimento de robôs móveis ao promover flexibilidade e reusabilidade do código. A portabilidade foi alcançada através da adoção do SO GNU/Linux, que suporta diferentes plataformas de hardware, e bibliotecas POSIX, garantindo que a plataforma seja executada em qualquer uma das distribuições que a implemente. Além disso, o reuso de código foi alcançado com sucesso, pois foi possível utilizar a classe *PositionStabilizer* para ambos os robôs 14-bis e Roburguer sem necessidade alterar o código fonte do controlador ou dos robôs.

Os experimentos mostram que a arquitetura de software adotada produz bons resultados, uma vez que o controlador adotado demonstrou o comportamento esperado com os dois robôs. Além disso, pode-se ver que a plataforma bem se adapta a ambientes estáticos. Para ser utilizada em ambientes dinâmicos, porém, é necessário que o usuário gerencie a atualização do mapa e do controle de posição. Desta forma, até o presente estágio de desenvolvimento, pode-se afirmar que a plataforma TAO pode ser usada em aplicações de mundo real com pequenos robôs em ambientes estáticos.

Quanto ao software do usuário, a plataforma TAO fornece os meios para abstrair alguns módulos físicos e lógicos do robô móvel, de forma que não é necessário que o usuário os crie novamente. Os módulos fornecidos também utilizam um *thread* (linha de execução) concorrente para que o usuário não precise atualizá-los – como ocorre com a classe *PositionStabilizer*. Isto facilitou o desenvolvimento do software do usuário por prover classes fáceis de usar que abstraem os módulos dos robôs.

Trabalhos futuros abordarão o refinamento das técnicas de controle de posição e trajetória para minimizar os erros. Outro foco de pesquisa futura é o desenvolvimento de um mecanismo de comunicação que permita aos robôs trocar informações.

6 Agradecimento

Esse trabalho foi apoiado pelo Laboratório de Automação Integrada e Robótica (LAIR) do Departamento de Projeto Mecânico, Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, e também pelo Laboratório de Integração de Sistemas e Dispositivos Inteligentes, Departamento de Computação, Faculdade de Ciências, Universidade Estadual Paulista. O trabalho foi financiado pela Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP) sob o processo 2010/02000-0.

Referências

1. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software (2009).
2. Gerkey, B., Vaughan, R.T., Howard, A.: The player/stage project: Tools for multi-robot and distributed sensor systems. Proceedings of the 11th international conference on advanced robotics. pp. 317–323 (2003).
3. Bruyninckx, H.: Open robot control software: the OROCOS project. Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on. pp. 2523–2528 (2001).
4. McCall, J.A.: Quality Factors. Encyclopedia of Software Engineering. John Wiley & Sons, Inc. (2002).
5. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software (2009).
6. Utz, H., Sablatnog, S., Enderle, S., Kraetzschmar, G.: Miro - middleware for mobile robot applications. Robotics and Automation, IEEE Transactions on. 18, 493–497 (2002).
7. Mok, S.M., Wu, C.: Automation integration with UPnP modules. Electronic Design, Test and Applications, 2006. DELTA 2006. Third IEEE International Workshop on. p. 5 pp. (2006).
8. Prata, S.: C++ Primer Plus. Sams Publishing, Indianapolis, Indiana (2004).
9. Linux Kernel Organization: The Linux Kernel Archives, http://www.kernel.org/.
10. Siegwart, R., Nourbakhsh, I.R.: Introduction to autonomous mobile robots. MIT Press (2004).
11. Alves, S.F. dos R., Ferasoli Filho, H., Pegoraro, R., Caldeira, M.A.C., Rosario, J.M., Yonezawa, W.M.: Proposal of Educational Environments with Mobile Robots. Proceedings of the 5th IEEE International Conference on Robotics, Automation and Mechatronics. pp. 264–269. , Qingdao, China (2011).
12. Siciliano, B., Khatib, O. eds: Springer Handbook of Robotics. Springer, Heidelberg (2008).
13. Bekey, G.A.: Autonomous robots: from biological inspiration to implementation and control. MIT Press (2005).
14. Hong, J., Park, K.: A new mobile robot navigation using a turning point searching algorithm with the consideration of obstacle avoidance. Int J Adv Manuf Technol. 52, 763–775 (2010).
15. Dudek, G., Jenkin, M.: Computational Principles of Mobile Robotics. Cambridge University Press (2010).
16. Ferasoli Filho, H., Caldeira, M.A.C., Alves, S.F. dos R., Valadão, C., Bastos Filho, T.F.: Use of Myoelectric Signals to Command Mobile Entertainment Robot by Disabled Children: Design and Control Architecture. Proceedings of the 3rd IEEE Biosignals and Biorobotics conference (ISSNIP). , Manaus (2012).