UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ABORDAGEM PARA CRIAÇÃO DE LINGUAGENS ESPECÍFICAS DE DOMÍNIO PARA ROBÓTICA MÓVEL

DANIEL BRUNO FERNANDES CONRADO

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos – SP

Junho/2012

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ABORDAGEM PARA CRIAÇÃO DE LINGUAGENS ESPECÍFICAS DE DOMÍNIO PARA ROBÓTICA MÓVEL

DANIEL BRUNO FERNANDES CONRADO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software

Orientador: Prof. Dr. Valter Vieira de Camargo

São Carlos – SP

Junho/2012

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

C754ac

Conrado, Daniel Bruno Fernandes.
Abordagem para criação de linguagens específicas de domínio para robótica móvel / Daniel Bruno Fernandes Conrado. -- São Carlos : UFSCar, 2012.
90 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2012.

1. Engenharia de software. 2. Modelagem. 3. Robôs móveis. 4. Processo de software orientado a objetos. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Abordagem para Criação de Linguagens Específicas de Domínio para Robótica Móvel"

Daniel Bruno Fernandes Conrado

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Prof. Dr. Valter Vieira de Camargo
(Orientador - DC/UFSCar)

Prof. Dr. Daniel Lucrédio
(DC/UFSCar)

Profa. Dra. Maria Istela Cagnin Machado
(UFMS)

São Carlos
Novembro/2012

Aos meus irmãos, Bruna e Juninho. Espero que vocês tenham muito mais sorte do que eu tive nessa vida e nunca se esqueçam das pessoas que passaram e passarão por suas vidas.

AGRADECIMENTOS

Agradeço a meu orientador, professor Valter, por ter me incentivado e dado todo o apoio e fé, mesmo quando eu não fazia por merecer isso. Agradeço-o também por estar sempre presente nos churrascos da turma e principalmente por ter me dado a chance de apresentar nosso trabalho em um congresso nos Estados Unidos. Se dependesse de mim, teria perdido essa experiência única.

Agradeço minha esposa, Yumi, a melhor esposa do mundo. Sempre esteve comigo, em todos os momentos, altos e baixos, terríveis e maravilhosos. A única coisa que tínhamos quando nos mudamos para São Carlos era um ao outro. Tudo que passamos juntos aqui provou o quão forte é o nosso amor.

Agradeço a minha família pelo carinho: Ronaldo, Juninho, Bruna, Gilson, Esther, Guinho e especialmente minha mãe, Dinha.

Agradeço aos meus amigos que conheci das turmas de mestrado que estiveram nesses anos aqui no DC. Todos eles foram muito importantes para minha formação tanto pessoal quanto profissional.

Agradeço a André Nicoleti e Rodrigo Barro por terem me ajudado no projeto, testado meu trabalho e fornecido as provas de conceito.

Agradecimentos especiais a meus mentores: Tsen Chung Kang, pastor Gilberto Stéfano, Zé Pedro Ibide, Stephen Kunihiro e Paulo Brabo.

Aquilo que mais ou menos aprendi é que escrevemos, falamos e lemos como se as coisas fossem sólidas, e a experiência insiste em demonstrar que são fluidas.

Paulo Brabo

RESUMO

Robôs móveis autônomos são máquinas com potencial para realizar atividades repetitivas ou de alta periculosidade com mais eficácia. Muitos possuem um software embarcado responsável pelo seu funcionamento. Nos últimos anos, a complexidade dessas aplicações robóticas embarcadas tem crescido continuamente e apresentam desafios que são incomuns ao desenvolvimento dos tradicionais sistemas de informação. Portanto, toda técnica que dê suporte a esse tipo de desenvolvimento pode contribuir significativamente. Uma técnica que permite o aumento de produtividade é a utilização de linguagens específicas de domínio (DSLs). Essas são linguagens de modelagem e programação cujas construções são conceitos e abstrações de um domínio de aplicação em particular. Isso desobriga o desenvolvedor de se preocupar com conceitos genéricos de programação (classes, objetos, atributos, etc.) para focar-se no problema a ser resolvido. Como o desenvolvimento de uma DSL não é uma tarefa trivial e tendo em vista as idiossincrasias dos robôs móveis autônomos, esta dissertação apresenta uma abordagem para construção de DSLs para robôs móveis. O objetivo é deixar mais sistemática e controlada a criação de DSLs para esse domínio. Nessa abordagem, uma aplicação é tomada como entrada e dela extraem-se declarações a respeito do domínio. Essas declarações são categorizadas e, para cada categoria, são levantadas partes comuns e variáveis. Então, essas partes são transformadas em componentes de uma DSL. Uma característica importante da abordagem apresentada é que uma versão inicial da DSL pode ser alcançada tendo apenas uma aplicação como base. Sugere-se que essa mesma DSL possa evoluir pela reaplicação da abordagem tendo uma nova aplicação como entrada. Dessa forma, novos componentes podem ser criados e os existentes, modificados. Também é apresentado um modelo genérico de linguagem que fornece uma arquitetura básica, permitindo que novas DSLs sejam facilmente construídas pela extensão da mesma. Duas provas de conceito são apresentadas com a intenção de exemplificar a aplicação da abordagem.

Palavras-chave: Linguagens Específicas de Domínio, Robôs Móveis, Engenharia de DSLs

ABSTRACT

Autonomous mobile robots are machines capable of executing repetitive/dangerous tasks more efficiently. Most of them have an embedded software which is responsible for their execution. Over the last years, the complexity of these applications has continuously growing and they are presenting challenges that are uncommon to traditional information systems' development. Therefore, any technique that can support their development is a great contribution. A technique that improves the productivity is to use domain-specific languages (DSLs). These are modeling and programming languages whose constructs are concepts and abstractions of a particular domain. It frees developers from worrying about generic programming concepts (classes, objects, attributes, etc.) and allows them to focus on the problem to be solved. As creating a DSL is not a trivial task and pointing the idiosyncrasies of mobile robots, this dissertation presents an approach for engineering DSLs to mobile robots. The aim is to make the activity of creating DSLs to this domain more systematic and controlled. In this approach, an application is taken as input and a series of domain statements is extracted from it. These statements are classified into categories and each one of them are analized in order to extract commonalities and variabilities, wich are transformed into components of a DSL. An important characteristic of the approach is that it asks for just one application to reach a first version of a running DSL. We suggest that the same DSL can be evolved just by applying the approach again using another application as input. So new components could be created and the existing ones could be modified. We also present a generic language model providing a foundation architecture that allows one to easily create new DSLs by extending it. Two proofs of concept are presented in order to exemplify the application of our approach.

Keywords: Domain-Specific Languages, Mobile Robots, DSL Engineering

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contexto

Sistemas embarcados são sistemas computacionais com propósitos específicos, geralmente feitos para serem fabricados em série e utilizarem o mínimo de recursos possível. Esses sistemas estão presentes em celulares, refrigeradores, rádios, injeções eletrônicas automotivas, calculadoras científicas, televisões, eletrodomésticos, entre outros. Alguns desses sistemas são *críticos*, significando que seu mal funcionamento pode colocar vidas em risco e/ou ocasionar grandes perdas financeiras. Um exemplo desse tipo de sistema são os braços robóticos soldadores de uma linha de montagem de carros; outros exemplos incluem controles de avião, dispositivos médicos, robôs móveis para exploração de terrenos vulcânicos, tratores e carros robotizados (WOLF, 2008).

Embora alguns sistemas embarcados possam ser desenvolvidos utilizando linguagens de programação de alto nível (por exemplo, Java, Objective-C e Python), muitos deles e principalmente os críticos são desenvolvidos com linguagens de baixo nível, como C e Assembly, em consequência das restrições a eles inerentes, como desempenho, memória reduzida e tolerância a falhas. No entanto, Liggesmeyer e Trapp (2009) afirmam que a complexidade de tais sistemas tem crescido rapidamente, e muitas empresas tem encontrado problemas com o seu desenvolvimento (criação, testes, manutenção, etc.), comprometendo a qualidade dos mesmos.

Como alternativa a esse desenvolvimento tradicional de sistemas embarcados, surge o Desenvolvimento Dirigido por Modelos (MDD – *Model-Driven Development*). Seu uso pode resultar em benefícios consideráveis, principalmente o aumento da produtividade, da facilidade de manutenção, do reuso de código e da qualidade dos sistemas. Essa técnica visa principalmente lidar com a crescente complexidade dos atuais sistemas oferecendo uma maneira de construí-

los que não dependa, por exemplo, do conhecimento de inúmeras linguagens de programação, bibliotecas ou plataformas; o desenvolvedor pode concentrar-se apenas no que é necessário.

Os sistemas são especificados por meio de modelos, que são mais abstratos e intuitivos que grandes quantidades de código escrito em linguagens de programação de propósito geral. Como o desenvolvimento do software ocorre a nível de modelos, há uma redução do *gap* semântico entre o domínio do problema e o domínio da solução ou implementação. Depois de prontos, esses modelos são transformados automaticamente em código-fonte por um gerador de código. No entanto, essa transformação automática só é possível quando tanto a linguagem de modelagem utilizada para construir os modelos quanto o gerador de código são restritos a um domínio em particular. Claramente não se pode construir um gerador universal, capaz de gerar qualquer aplicação. De forma similar, é difícil uma linguagem de modelagem de propósito geral como a UML mapear-se a um domínio de problema de tal maneira que possa tornar realidade a geração *completa* de código. Nesse sentido, para aumentar o nível de abstração no MDD, as linguagens de modelagem precisam estar mais alinhadas ao domínio (KLEPPE; WARMER; BAST, 2003).

Linguagens que são direcionadas à resolução de um determinado problema são chamadas de Linguagens Específicas de Domínio (*Domain-Specific Languages*–DSL). Dentre as mais conhecidas, estão as linguagens de projeto e consulta de bancos de dados e linguagens de projeto de interface de usuário. É possível ter DSLs em domínios horizontais (técnicos), como persistência, comunicação ou transações, e até mesmo domínios verticais (negócios), como telecomunicações, bancário, controle robótico ou seguros; elas permitem escrever aplicações apenas com expressões relativas a esses domínios. Outra característica das DSLs é que podem ser tanto textuais quanto gráficas, e que podem inclusive fazer ou não parte de uma outra linguagem (como uma espécie de especialização). As DSLs que especializam outras linguagens são chamadas de DSLs internas, ou embarcadas, e restringem-se à sintaxe e outras particularidades da linguagem hospedeira. As DSLs que possuem sua própria sintaxe e semântica são chamadas de DSLs externas e geralmente exigem um esforço maior para serem criadas, demandando softwares de apoio ao seu desenvolvimento. Além disso, a criação de uma DSL não é um processo trivial, principalmente quando o domínio é de tal complexidade como o de sistemas críticos (KELLY; TOLVANEN, 2008b).

1.2 Motivação e Objetivos

Há muitos trabalhos na literatura que fornecem modelos de processos e diretrizes que auxiliam na criação de DSLs (MERNIK; HEERING; SLOANE, 2005; EVERMANN; WAND, 2005; ROBERT

et al., 2009; STREMBECK; ZDUN, 2009; GÜNTHER; HAUPT; SPLIETH, 2010) e que estão focados ou no domínio de sistemas de informação ou em nenhum domínio especificamente. Como os sistemas embarcados possuem idiossincrasias bem incomuns aos tradicionais sistemas de informação, o objetivo deste trabalho é facilitar o desenvolvimento de DSLs para esse domínio pelo fornecimento de uma abordagem com diretrizes específicas para a sua construção; diretrizes que estão voltadas para um tipo de sistema embarcado em particular: os robôs móveis.

Similarmente ao processo de desenvolvimento de software, todos esses processos de criação de DSLs possuem em comum as seguintes atividades: análise, projeto, implementação e integração. Além disso, há basicamente três modos de conduzir o processo (STREMBECK; ZDUN, 2009):

- Dirigido pelo modelo da linguagem: o processo começa pela definição do modelo da DSL seguida de sua sintaxe concreta (sua representação gráfica), de seu comportamento e, por fim, de seu mapeamento à plataforma de execução;
- Linguagem maquete (*mockup language*): o desenvolvimento da DSL começa pela definição de sua sintaxe concreta ou notação visual, com a participação dos *experts* do domínio, e depois as demais partes da DSL são derivadas dessa notação;
- Extração de uma DSL a partir de um sistema existente: as abstrações do domínio que podem compor uma DSL são derivadas diretamente de algum software pronto e também de quaisquer documentações que esse software possa ter.

A abordagem proposta neste trabalho leva em consideração esse último tipo de processo, ou seja, uma aplicação robótica já existente é tomada como entrada. Esse tipo foi escolhido justamente porque envolve a criação de código-fonte, e como esse é um domínio incomum para o autor, foi considerado que a experiência de criar aplicações desse domínio contribuiria muito para o entendimento do mesmo e não somente a leitura de documentos e reuniões com *experts*.

Essa abordagem surgiu pela experiência obtida no desenvolvimento de uma DSL a partir de um robô monitorador de portas. Para realizar a análise, é fornecido um modelo para estruturar os requisitos da aplicação de entrada em formato de árvore, em que os níveis possuem distintos graus de abstração e os nós possuem uma relação causa-efeito. Esse modelo failita a identificação de elementos da DSL e permite selecionar o seu grau de flexibilidade e expressividade.

Além disso, foi desenvolvido um modelo genérico que pode ser utilizado como base para criar DSLs voltadas à robótica. Esse modelo possui entidades e relacionamentos abstratos que implementam uma arquitetura de subsunção (BROOKS, 1986), que é uma forma de implemen-

tação da inteligência de um robô. O desenvolvedor estende as entidades e relacionamentos do modelo genérico e especifica, por exemplo, quais são os comportamentos que a DSL em questão oferecerá suporte. Como prova de conceito, foi desenvolvida uma DSL utilizando as especificações das atividades supracitadas tendo como base a aplicação de um robô que realiza a entrega de materiais entre as salas de um ambiente (por exemplo, o robô pode realizar a entrega de remédios e instrumentos médicos entre quartos de um hospital).

1.3 Organização

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta os principais conceitos utilizados nesta dissertação; o Capítulo 3 apresenta os trabalhos relacionados; o Capítulo 4 apresenta a abordagem proposta juntamente com um exemplo de sua aplicação na construção de uma DSL; o Capítulo 5 apresenta a prova de conceito e o Capítulo 6 conclui a dissertação.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordados os principais conceitos envolvidos na confecção deste trabalho. Este compreende três seções: a seção 2.1 introduz os sistemas embarcados, a seção 2.2 apresenta os robôs móveis e a seção 2.3 apresenta o Desenvolvimento Dirigido Por Modelos.

2.1 Sistemas Embarcados

Sistemas embarcados são sistemas computacionais que executam poucas ou apenas uma função dedicada *dentro* de um equipamento eletromecânico (WOLF, 2008). Para executar essas funções, eles utilizam microprocessadores de propósito geral juntamente com hardware especializado. Um forno de microondas, por exemplo, possui um sistema embarcado que controla o motor do prato de vidro giratório, a intensidade das microondas e o painel de funções.

Os sistemas embarcados surgiram em meados da década de 70, quando surgiu o microprocessador. Os sistemas de controle da época que utilizavam lógica digital eram implementados com componentes lógicos (por exemplo, portas AND, OR, etc); um exemplo são as calculadoras da época, em que utilizavam-se centenas de componentes para implementar uma simples calculadora de quatro funções. Esses componentes foram posteriormente embutidos em circuitos integrados, criando funcionalidades de mais alto nível. Com isso, foi possível criar um único circuito integrado contendo todas as funções de uma calculadora, reduzindo o custo para fabricar esses sistemas. Porém, para qualquer mudança ou nova funcionalidade requerida, era necessário construir um novo circuito integrado e o processo de construção era relativamente demorado. Para resolver esse problema, surgiu o microprocessador: um circuito que lê instruções e as processa. Adicionar uma nova funcionalidade resume-se simplesmente em alterar essas instruções (HEATH, 2003).

Projetar sistemas embarcados é uma atividade complexa porque envolve não só problemas de software mas também de mecânica, pneumática, elétrica, entre outros. O fato dos sistemas embarcados serem programados em microprocessadores é uma vantagem em relação aos circuitos integrados porque o hardware pode ser reutilizado, ou seja, criar uma nova aplicação ou adicionar uma funcionalidade a uma aplicação existente pode resumir-se simplesmente em alterar o software que roda no microprocessador, enquanto que, se o sistema fosse feito utilizando circuitos integrados, seria necessário redesenhá-los e passá-los pelo processo de fabricação, demandando muito mais tempo. Com isso, é possível criar facilmente famílias de produtos (WOLF, 2008). Por exemplo, pode-se criar uma família de fornos de microondas, em que cada forno possui um menu diferente, exigindo um software para cada menu, porém, pode-se utilizar o mesmo microprocessador e as mesmas peças internas do forno. A variabilidade dessa família está no painel de funções e no software; o hardware permanece intacto. É mais rápido e barato que projetar circuitos integrados para cada forno.

Uma das principais características dos sistemas embarcados é o conjunto de restrições que eles devem atender, as quais são normalmente incomuns em sistemas para computadores de propósito geral. Eles compreendem algoritmos sofisticados, que envolvem processamento de imagens e cálculos complexos com grandes conjuntos de variáveis; precisam controlar interfaces de usuário complexas (por exemplo, navegação em GPS[1]), possuem restrições de tempo real e se não responderem dentro de um intervalo de tempo determinado, podem deixar clientes insatisfeitos (por exemplo, demorar demais para transmitir voz em uma ligação de celular) ou mesmo colocar vidas em risco (por exemplo, um marcapasso não detectar a tempo uma queda nos batimentos cardíacos). O custo de fabricação deve ser viável e isso é afetado por vários fatores, como o tipo de processador e os dispositivos de entrada e saída a serem usados, bem como o quanto os clientes podem pagar pelo produto final. Além disso, deve-se levar em consideração a quantidade de energia requerida pelo sistema, que afeta o tempo de vida da bateria e também influencia na dissipação de calor (WOLF, 2008).

Os sistemas embarcados podem ser classificados em três categorias, de acordo com o seu poder de processamento: sistemas domésticos e de escritório, sistemas de médio porte e sistemas de alto desempenho. Os sistemas domésticos e de escritório precisam combinar bom desempenho, baixo preço e baixo consumo de energia; por isso, geralmente utilizam microprocessadores de 4 ou 8 bits e têm boa parte de seus códigos implementados diretamente em *assembly* para que fiquem pequenos e ocupem pouca memória. Exemplos dessa categoria são telefones fixos e refrigeradores (WOLF; FREY, 1992).

[1] *Global Positioning System* – Sistema de Posicionamento Global

Os sistemas de médio porte realizam funções mais sofisticadas e complexas porém ainda necessitam ser de baixo custo. Esses sistemas utilizam processadores de 16 ou 32 bits e possuem muito código; é comum o emprego de linguagens de programação de alto nível, mas o tamanho do código deve ser considerado, visto que geralmente são armazenados em memórias ROM[2] (WOLF; FREY, 1992).

Os sistemas de alto desempenho são geralmente multiprocessados (i.e. utilizam mais de um microprocessador) e possuem restrições de tempo real críticas. Processamentos que exigem altas velocidades podem ser implementados com hardware específico/personalizado ou microprocessadores com códigos bem otimizados. Exemplos dessa categoria são aplicações de processamento de sinais, como sistemas de telecomunicações, e sistemas de controle de tráfego aéreo (WOLF; FREY, 1992).

2.2 Robôs Móveis

Robôs são máquinas eletromecânicas que atuam no ambiente em que estão inseridas geralmente para realizar tarefas repetitivas ou que ofereçam riscos às pessoas. Os robôs extraem informações desse ambiente por meio de *sensores* e atuam no mesmo por meio de *atuadores*. Eles possuem uma unidade de controle responsável por processar os dados obtidos pelos sensores e acionar os atuadores conforme a funcionalidade a que foram programados. Essa unidade, antigamente, era desenvolvida em grandes e caros computadores, aos quais eram conectados os sensores e atuadores por meio de cabos ou comunicação sem fio. Hoje em dia, é possível programar a funcionalidade dos robôs em pequenos e baratos sistemas embarcados – isso permitiu maior mobilidade aos robôs. A palavra *robô* vem de *robota* que, em tcheco, significa *"trabalho forçado"* (ARKIN, 1998; SIEGWART; NOURBAKHSH, 2004; BRÄUNL, 2006). Os robôs são largamente utilizados na indústria automotiva; na linha de produção, são braços móveis que soldam os componentes do carro ou realizam a pintura da carroceria.

Os robôs móveis mais simples são aqueles que se locomovem por rodas. Na Figura 2.1, tem-se três possíveis configurações de rodas que podem ser usadas em robôs. Todo robô com rodas possui uma ou mais *rodas de propulsão (ou tração)* (*driven wheels*), representadas por retângulos preenchidos, zero ou mais *rodas passivas* (*passive wheels*), representadas por retângulos não preenchidos e zero ou mais *rodas de direção* (*steering wheels*), representadas dentro de um círculo. As rodas de propulsão dão movimento ao robô, as rodas passivas servem para sustentação e as rodas de direção servem para guiar o robô. São acoplados motores nas rodas

[2] *Read-Only Memory* (Memória Somente de Leitura)

de propulsão e de direção. Na configuração de rodas da parte (a) da figura, a roda de propulsão é também de direção e, portanto, possui dois motores, um para girá-la em torno de si mesma (propulsão) e outro para girá-la para os lados (direção) (BRÄUNL, 2006).

A configuração de rodas da parte (b) da Figura 2.1 é a mais utilizada e é chamada de *direção diferencial* (*differential drive*). As vantagens que ela tem sobre as demais são a possibilidade de girar o robô em torno de si mesmo e a de não precisar girar as rodas para fazer curvas, diminuindo consideravelmente a complexidade do robô. A configuração de rodas da parte (c) é chamada de "Direção de Ackermann" (*Ackermann Steering*). Há dois motores: um para as duas rodas de propulsão e outro para as duas rodas de direção. Essa configuração é a padrão para a maioria dos carros de passeio.

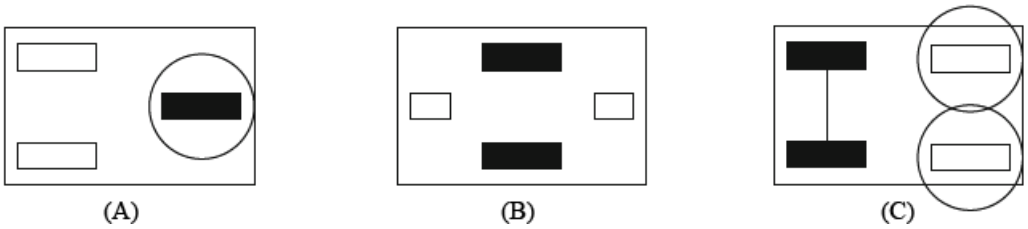
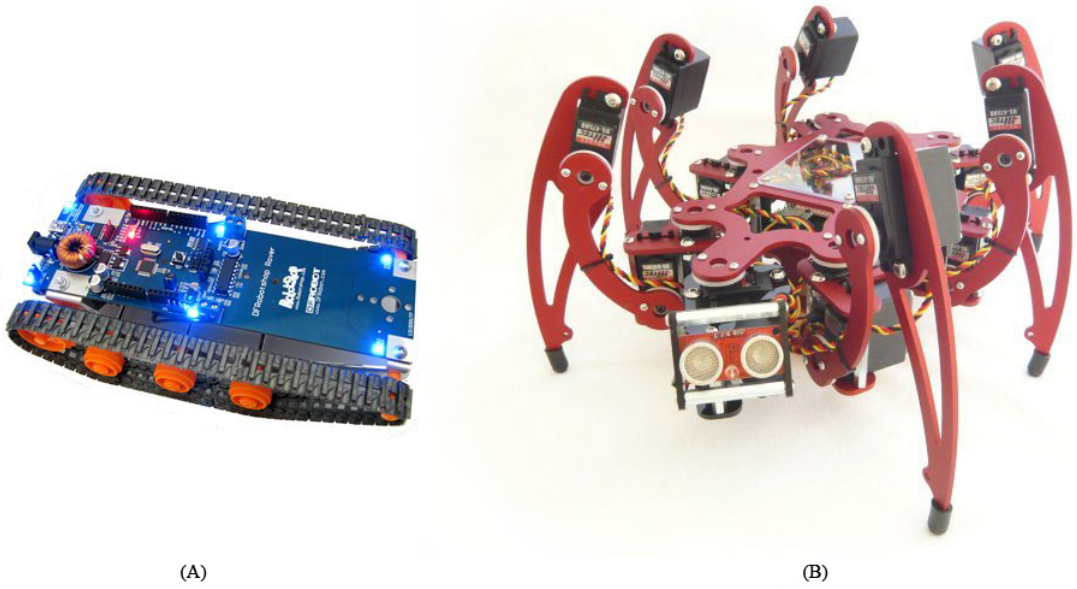


Figura 2.1: Tipos de robôs com rodas (BRÄUNL, 2006)

Além das rodas, os robôs também podem se locomover por esteira (Figura 2.2 à esquerda) e por pernas (Figura 2.2 à direita). Uma vantagem que a esteira tem sobre as rodas é a facilidade de se locomover em superfícies irregulares, embora a navegação fique prejudicada. Já robôs com pernas não só podem andar em superfícies irregulares como também podem subir e descer degraus. De modo geral, quanto mais pernas o robô tiver, mais fácil será para mantê-lo estável. Por exemplo, o robô de seis pernas que está na parte direita da Figura 2.2 pode se locomover mantendo sempre três pernas no chão, formando um tripé (duas de um lado e uma do outro) e garantindo a estabilidade. Essa técnica, no entanto, não pode ser aplicada em robôs de duas pernas.

2.2.1 Sensores

Os robôs utilizam sensores para conhecer o ambiente em que estão inseridos e também para conhecer a si mesmo. Segundo Groover (apud ARNOLD, 2007, p. 10), os sensores podem ser divididos em duas categorias: internos e externos. Os sensores internos são utilizados para autoconhecimento, como sua posição e velocidade atual. Os sensores externos são utilizados para obter informações do ambiente com o objetivo de coordenar a atuação do robô no mesmo. os tipos de sensores externos podem ser: táteis, de proximidade, de visão, entre outros.



(A) (B)

Figura 2.2: Um robô com esteira e outro com seis pernas. Adaptado de (ROBOTSHOP, 2011; DVICE, 2011)

Os sensores táteis detectam quando há um contato entre eles e um objeto. Alguns sensores táteis podem medir a força do contato. Sensores de proximidade medem a distância em que estão de objetos. Duas tecnologias utilizadas nesses sensores são ultrassom e *laser*; esse tipo de sensor geralmente é chamado de *range sensor*. Sensores de visão podem ser utilizados para inspecionar objetos e guiar o robô. Existem ainda outros sensores que podem ser usados, como sensores de temperatura, de tensão e corrente elétrica, vazão e pressão de fluidos.

2.2.2 Veículos de Braitenberg

Braitenberg (1984) descreve uma abstração conceitual de robôs móveis em que, dependendo da configuração de seus sensores e atuadores, demonstram comportamentos semelhantes à covardia, agressividade, amor e otimismo. Um dos exemplos apresentados mostra um robô com dois sensores de luz e duas rodas. Na Figura 2.3, o robô é representado pelo retângulo não preenchido, as rodas são representadas pelos retângulos preenchidos e os sensores são representados por círculos. Cada sensor está conectado à roda que está do mesmo lado. O sensor aumenta a velocidade da roda na mesma proporção em que recebe luz. Ao colocar uma fonte de luz na frente dos sensores, o robô se movimentará em direção a ela. Porém, se o sensor esquerdo começar a receber mais luz que o direito, em consequência da falta de alinhamento entre o robô e a fonte de luz, a roda esquerda andará mais rápido, fazendo com que o robô desvie da fonte de luz (c.f. Figura 2.3).

Outro exemplo, apresentado na Figura 2.4, é um robô em que o sensor de luz esquerdo está

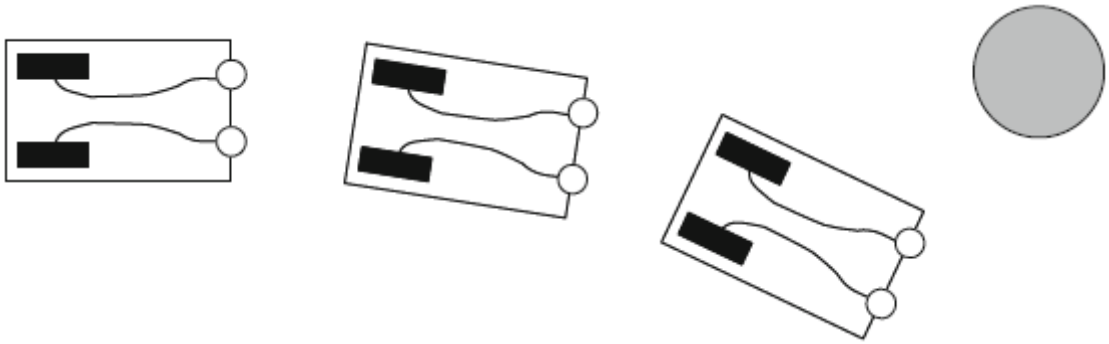


Figura 2.3: Robô que desvia da fonte de luz (BRÄUNL, 2006)

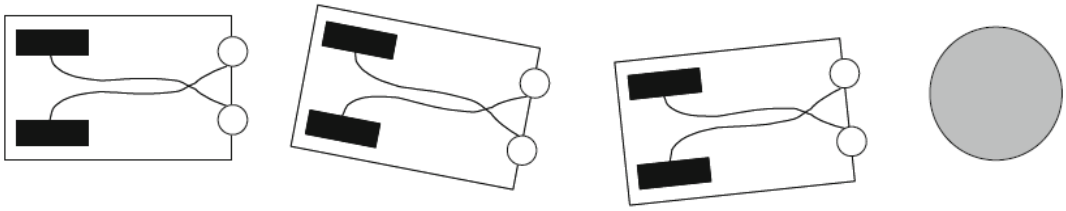


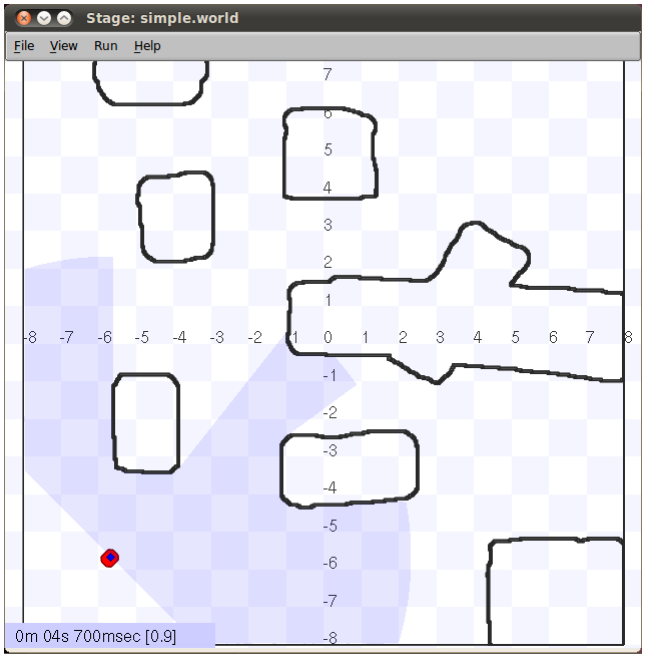
Figura 2.4: Robô que anda em direção à luz (BRÄUNL, 2006)

conectado à roda direita e o sensor direito à roda esquerda. Se os sensores recebem a mesma intensidade de luz, o robô possui o mesmo comportamento que o anterior. Porém, se o sensor esquerdo recebe mais luz, a roda direita se move mais rápido e faz o robô andar em direção à luz. Ao observar esses dois robôs por um tempo, percebe-se que ambos têm aversão à luz mas, enquanto o primeiro a evita, o segundo corre em sua direção e consequentemente colide-se nela. Pode-se notar, portanto, que o primeiro demonstra sentimento de *covardia*, enquanto o segundo, *agressividade*.

As próximas subseções apresentam três plataformas de software para desenvolvimento de aplicações robóticas: *Player Project*, *Microsoft Robotics Studio* e *LeJOS*.

2.2.3 *Player Project*

O projeto *Player*, o qual é formalmente chamado de *The Player/Stage/Gazebo Project* (PLAYER/STAGE/GAZEBO..., 2011), é um conjunto de três softwares livres (*Player*, *Stage* e *Gazebo*) que oferecem um ambiente de programação para robôs móveis. O *Player* é a peça fundamental do ambiente; ele é um servidor que oferece uma interface de conexão TCP (*Transmission Control Protocol*) para os sensores e atuadores do robô. Por causa disso, a funcionalidade do robô pode ser programada em praticamente qualquer linguagem de programação com suporte a TCP. Um mesmo código também pode controlar mais de um robô e um mesmo robô pode ser controlado por mais de um código concorrentemente e em diferentes máquinas, o que permite



(a)

(b)

(c)

Figura 2.5: Simulador Stage

controlar os sensores e atuadores de forma colaborativa e distribuída.

O *Stage* é um software de simulação em 2D e trabalha com o *Player* de modo transparente, ou seja, o *Player* não precisa ter conhecimento de que está em simulação. O *Stage* fornece uma maneira de construir robôs virtuais e suporta vários tipos de sensores e atuadores, como sensores de proximidade que utilizam *laser* ou ultrassom, câmeras com zoom e visão panorâmica, e sensores de odometria[3]. Com o *Stage*, é possível criar mundos virtuais, com paredes e objetos de várias formas. A especificação do mundo virtual é feita em um arquivo *world*. Nesse arquivo, é possível especificar o tamanho da janela de simulação, o zoom, a velocidade de simulação, bem como o formato dos robôs e a localização dos sensores nos mesmos. Na Figura 2.5, parte 2.5a, tem-se uma janela de simulação do *Stage*. A simulação contém o robô Pioneer 2 DX (parte 2.5b) com o sensor *laser* SICK LMS200 (parte 2.5c). Essa combinação robô-sensor é comumente utilizada para fins de pesquisa.

O *Gazebo*, assim como o *Stage*, também é um software de simulação que oferece todos os recursos supramencionados; a principal diferença é que a simulação é em 3D. O *Gazebo*

[3]Estimação da posição do robô no tempo

Figura 2.6: Brick NXT da LEGO Mindstorms (KNUTH, 2011)

oferece um *feedback* realista dos sensores e também simula a física dos corpos com precisão. Enquanto o *Stage* é utilizado para simular a interação de vários robôs com pouca precisão, o Gazebo foi feito para simular uma pequena quantidade de robôs com muita precisão. E como eles são compatíveis com o *Player*, é possível que o mesmo código seja simulado nos dois ambientes com pouca ou nenhuma modificação. O usuário, portanto, pode utilizá-los de forma complementar.

2.2.4 Microsoft Robotics Studio

Microsoft Robotics Studio (MRS) (MICROSOFT, 2011b) é um ambiente de desenvolvimento de aplicações para robótica e é destinado tanto a ambientes acadêmicos quanto corporativos. Permite criar e simular aplicações de robótica facilmente e oferece suporte a uma vasta variedade de hardware. O MRS fornece uma Linguagem Visual de Programação (*Visual Programming Language* – VPL) que teoricamente permite que não-programadores criem suas próprias aplicações, por meio da conexão de blocos que representam serviços; é possível também reutilizar uma coleção de blocos conectados como um único bloco em qualquer lugar do programa. O MRS oferece um ambiente de simulação 3D que também simula a física do mundo real. Além da VPL, o MRS oferece acesso aos sensores e atuadores por meio de uma biblioteca baseada em .NET. Isso permite que as aplicações sejam escritas em várias linguagens, como C#, Visual Basic, JScript e IronPython.

2.2.5 LeJOS

LeJOS (LEJOS, 2011b) é uma máquina virtual Java adaptada para os *bricks* programáveis (ver Figura 2.6) do kit LEGO Mindstorms, um brinquedo educativo, que é um conjunto de peças e dispositivos, como sensores e motores, para a construção de robôs (LEGO, 2011). A máquina virtual LeJOS acompanha uma biblioteca que contêm interfaces de alto nível para os sensores e atuadores do Mindstorms.

Além do LeJOS, o Lego Mindstorms também pode ser programado em diversas outras

linguagens. Um exemplo é o ambiente de desenvolvimento NXT-G que acompanha o kit e que tem por base o software LabVIEW (NI, 2011), que oferece uma linguagem gráfica para programar sistemas de medição, teste e de controle.

O LeJOS possui um modelo de programação de robôs com base em comportamentos (*Behavior Control Model*), que é uma modificação do modelo proposto por Brooks (1990). Esse modelo é uma alternativa à programação de robôs que é feita em um *loop* infinito contendo vários comandos de condição (`if's` e `else's`). Códigos feitos com esse tipo de programação são difíceis de entender e de expandir. O modelo de comportamentos do LeJOS é um modelo orientado a objetos que possui dois componentes principais: a classe *Arbitrator* (Árbitro) e a interface *Behavior* (Comportamento). Basicamente, o desenvolvedor escreve os comportamentos que seu robô deve possuir e envia-os ao árbitro para que ele regule quais comportamentos devem ser ativados. A interface *Behavior* possui três métodos:

- *takeControl()*: retorna um valor booleano indicando se o comportamento deve ser ativado ou não. Por exemplo, o comportamento ficará ativo quando um sensor de toque for pressionado;
- *action()*: inicia a ação que o robô deve fazer (por exemplo, girar o robô em 45º para a direita);
- *suppress()*: esse método deve parar as ações que estão ocorrendo no método *action()*.

Depois de passados os comportamentos como um vetor para o árbitro, ele executa o método *takeControl()* de cada comportamento até que um deles retorne verdadeiro. Ele, então, executa o método *action()* desse comportamento. Se dois comportamentos quiserem ser ativados ao mesmo tempo, apenas o de maior prioridade, que é proporcional ao seu índice no vetor, é executado. Os comportamentos que estão em execução são suprimidos (o método *suppress()* é chamado). Na Listagem 2.1, tem-se um exemplo do ponto de entrada de uma aplicação utilizando esse modelo. Na linha 4 é criado um vetor de comportamentos com duas posições, sendo que a primeira recebe o comportamento *Andar* (linha 5) e a segunda, o comportamento *DesviarObstaculo* (linha 6). Após, é criado um árbitro e passado a ele os comportamentos; ele, por fim, é iniciado.

Embora esse modelo de programação exija um pouco mais de planejamento antes de escrever o código, cada comportamento fica bem encapsulado em uma estrutura relativamente simples de entender e permite que comportamentos sejam adicionados ou retirados sem que isso repercuta negativamente para o restante do código (LEJOS, 2011a).

Listagem 2.1: Exemplo de um programa principal em LeJOS

```
1  public class Robo {
2    ...
3    public static void main(String[] args) {
4      Behavior[] comportamentos = new Behavior[2];
5      comportamentos[0] = new Andar();
6      comportamentos[1] = new DesviarObstaculo();
7      Arbitrator arbitro = new Arbitrator(comportamentos);
8      arbitro.start();
9    }
10 }
```

2.3 Desenvolvimento Dirigido por Modelos

Desenvolvimento Dirigido por Modelos ou *Model-Driven Development* (MDD), também chamado de *Model-Driven Software Development* (MDSD) e *Model-Driven Engineering* (MDE) é uma técnica de desenvolvimento de software que considera os modelos do software como principais artefatos de desenvolvimento. A principal diferença entre essa técnica e os processos típicos de desenvolvimento é que os modelos não são apenas "papéis", mas são o software de fato. Com essa técnica, o desenvolvedor não escreve código, apenas constroi modelos – o código é automaticamente gerado a partir deles (KLEPPE; WARMER; BAST, 2003; KELLY; TOLVANEN, 2008b).

Essa técnica de construção de software surgiu para tentar resolver alguns problemas que são recorrentes em processos de desenvolvimento de software. Segundo Kleppe, Warmer e Bast (2003), alguns dos problemas mais importantes são: produtividade e portabilidade. A produtividade é afetada pela criação de documentos e modelos do software dentro de um típico processo de desenvolvimento. Esses modelos são úteis apenas nas fases iniciais de desenvolvimento, em que são feitas reuniões entre as pessoas envolvidas no projeto para discuti-los e, assim, melhorar o entendimento do problema. A utilidade dos modelos, porém, diminui à medida em que a codificação avança. Isso acontece porque, durante o desenvolvimento, é comum ocorrerem mudanças no software, e essas (quase sempre) são implementadas diretamente no código, tornando os modelos criados obsoletos. Isso acontece porque atualizar os modelos não é uma atividade que contribui para produzir o software. De fato, o software é o código escrito e não os modelos. Mesmo eliminando a etapa de modelagem, permanece outro problema que diminui a produtividade, o qual está relacionado ao tempo empregado para que a equipe de manutenção entenda o software desenvolvido. Sem os modelos, os mantenedores têm à disposição somente o código que, na maioria dos casos, é denso e extenso.

O problema da portabilidade está na constante adaptação das empresas às novas tecnologias que surgem rapidamente da indústria de software. Essa adaptação é necessária porque tais

tecnologias são requeridas pelos clientes e resolvem problemas reais. Além disso, a indústria geralmente deixa de oferecer suporte a velhas tecnologias e a versões anteriores de ferramentas e bibliotecas. Consequentemente, as empresas se veem obrigadas a portar seus softwares para as tecnologias atuais, e todo o investimento feito em tecnologias anteriores (por exemplo, treinamentos e bibliotecas próprias) pode se tornar inútil.

Uma característica do MDD é que é possível ter diferentes níveis de modelos, sendo que há transformações de modelos entre os níveis. Essas transformações são comumente chamadas de *Model-To-Model* (M2M). O tipo de transformação que converte modelos em código é geralmente chamado de *Model-To-Text* (M2T).

Geralmente, é utilizada uma ferramenta para criar os modelos e uma ferramenta para criar as regras de transformação de modelos em código ou em outros modelos. Existem vários softwares que criam ferramentas de modelagem, dentre eles o *Graphical Modeling Project* (GMP) (GRAPHICAL..., 2011), *MetaEdit+* (METACASE..., 2011) e *Microsoft Visualization and Modeling SDK* (sucessor do *Microsoft DSL Tools*) (VISUAL..., 2011). As linguagens de modelagem, também chamadas de metamodelos, são comumente definidas com uma metalinguagem ou metametamodelo. A UML (*Unified Modeling Language*) (OMG..., 2011a), por exemplo, é um metamodelo especificado pelo metametamodelo *MetaObject Facility* (MOF) (OMG..., 2011b).

Nas duas seções seguintes são apresentadas duas abordagens de MDD: *Model-Driven Architecture* (MDA) e *Domain-Specific Modeling* (DSM).

2.3.1 *Model-Driven Architecture* (MDA)

Model-Driven Architecture é uma arquitetura de MDD desenvolvida pelo OMG (MDA, 2011). Ela define três níveis de modelos: Modelo Independente de Computação (*Computation Independent Model* – CIM), Modelo Independente de Plataforma (*Platform-Independent Model* – PIM) e Modelo Específico de Plataforma (*Platform-Specific Model* – PSM). Na Figura 2.7, tem-se uma ilustração das interações existentes entre esses modelos; as setas sólidas representam transformações e as setas tracejadas representam adição de informações adicionais para dar suporte às transformações.

O CIM é o modelo de mais alto nível de abstração; nele, não é considerada nenhuma característica computacional (por exemplo, estrutura de dados) e os conceitos do domínio são utilizados diretamente. Ele pode ser transformado em um PIM, cuja especificação tem um nível de abstração um pouco mais baixo, contendo elementos computacionais; no entanto, o PIM é genérico o suficiente para poder ser reutilizado em diferentes plataformas de implementação.

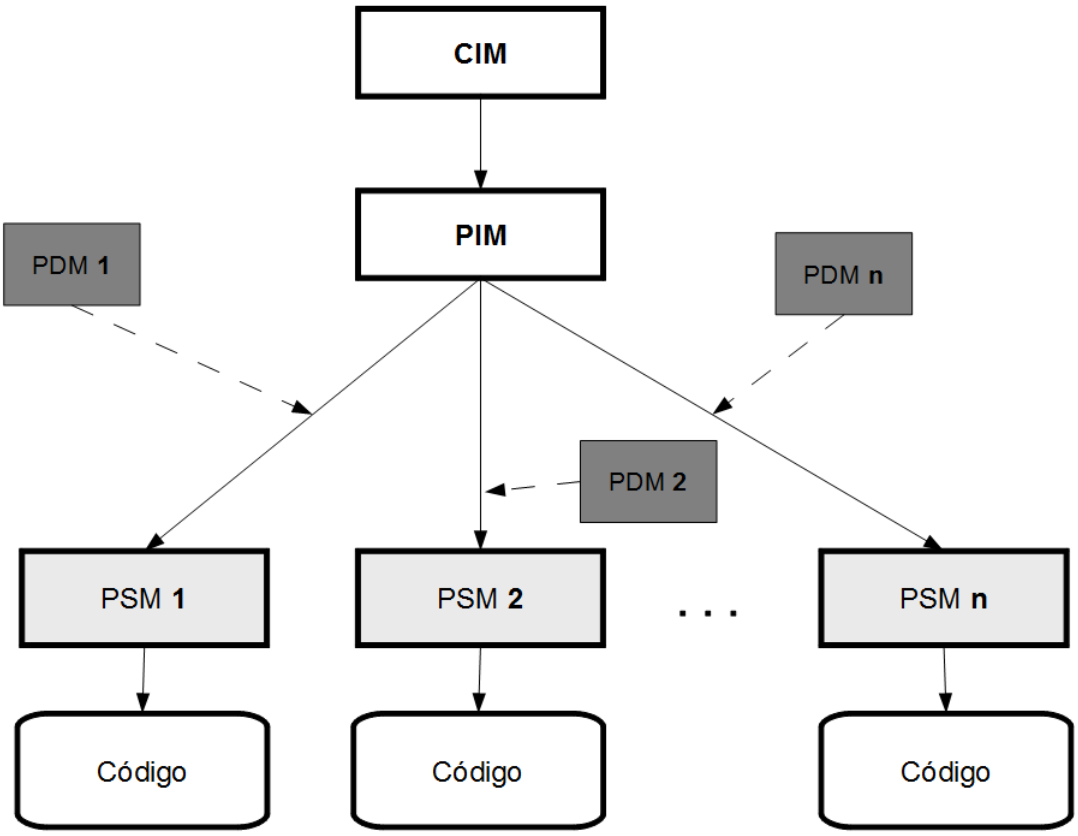


Figura 2.7: MDA – Visão geral

Apesar de ser possível automatizar a transformação de CIM para PIM, é uma atividade problemática porque envolve interpretação de requisitos e decisões de projeto.

O PIM pode ser transformado em vários PSMs, cada um para uma plataforma de implementação diferente. O PSM é um modelo que complementa as informações do PIM com detalhes específicos de uma plataforma e, a partir dele, pode-se gerar código automaticamente. A transformação de PIM para PSM pode ter por base um *Platform Definition Model* (Modelo de Definição de Plataforma – PDM), que pode corresponder, por exemplo, à plataforma *Common Object Request Broker* (CORBA) (OMG, 2011), JavaServer Faces (JSF) (ORACLE, 2011), .NET (MICROSOFT, 2011a), entre outras.

O metametamodelo utilizado na MDA é o MOF; com ele, é possível criar várias linguagens de modelagem. Os metamodelos (e modelos) feitos com MOF são descritos em um formato de XML chamado XMI (XML *Metadata Interchange*) (XMI, 2011), o que permite a interoperabilidade entre ferramentas de modelagem e de transformação de modelos. Embora seja possível criar novas linguagens com o MOF, a MDA tem grande foco na UML, que pode ser estendida por meio de *perfis* (*Profiles*), os quais adicionam informações específicas de plataforma que podem ser utilizadas para criar PSMs.

2.3.2 Modelagem Específica de Domínio

Modelagem Específica de Domínio (DSM - *Domain-Specific Modeling*) é uma abordagem de MDD cujo nível de abstração proporcionado é mais alto que o das linguagens de modelagem de propósito geral porque a especificação da solução é feita por meio de uma Linguagem Específica de Domínio (DSL – *Domain-Specific Language*) que utiliza diretamente os conceitos do domínio do problema. O produto final é gerado automaticamente a partir dessa especificação. A geração de código é feita por um Gerador (*Generator*), o qual é análogo a um compilador. Ele é responsável por interpretar um programa escrito com a DSL e gerar o código em uma ou mais linguagens de programação (por exemplo, Java e XML), e esse pode ter por base um *framework* do domínio (*Domain Framework*), que é basicamente uma biblioteca com funções específicas de domínio. Em síntese, uma plataforma de DSM constitui-se de três partes: uma linguagem, um gerador e um framework, todos específicos de um domínio (KELLY; TOLVANEN, 2008b).

A DSM pode ser aplicada não só em domínios técnicos, denominados horizontais (por exemplo, persistência, sincronização e segurança), como também em domínios de negócio, denominados verticais (por exemplo, telecomunicações, bancos e automação industrial). Geralmente, o tamanho do domínio de uma plataforma de DSM é pequeno, bem focado, e está relacionado a um determinado ambiente, plataforma ou produto, o que permite gerar código completo e funcional. Em sistemas complexos é comum o emprego de mais de uma plataforma de DSM, cada uma responsável por uma parte do software. Por exemplo, para desenvolver um software de banco pode-se ter uma plataforma de DSM para o domínio de empréstimos, outra para o domínio de investimentos e ainda outra para o domínio horizontal de concorrência (KELLY; TOLVANEN, 2008b).

Todo o esforço necessário para criar uma plataforma de DSM pode ser dividido entre seus três componentes: a DSL, o gerador e o framework de domínio. Isso quer dizer que, dependendo do domínio, o projetista pode obter a abstração desejada apenas criando uma DSL e um gerador trivial, que simplesmente mapeia as construções da DSL com pedaços fixos de código. Esse caso é ilustrado na parte (a) da Figura 2.8, na qual também estão outras três possibilidades. Na parte (b), o gerador já é um pouco mais complexo; por exemplo, ele pode conter análises de fluxo de controle e de dados. Na parte (c), alguns códigos responsáveis, por exemplo, por configurações da plataforma e funções auxiliares fazem parte do framework de domínio e o código gerado os utiliza. Já na parte (d) o framework de domínio é mais completo, contendo componentes reusáveis e serviços que não estão presentes na plataforma.

Apesar de haver várias maneiras de distribuir o trabalho entre os componentes, nem todas

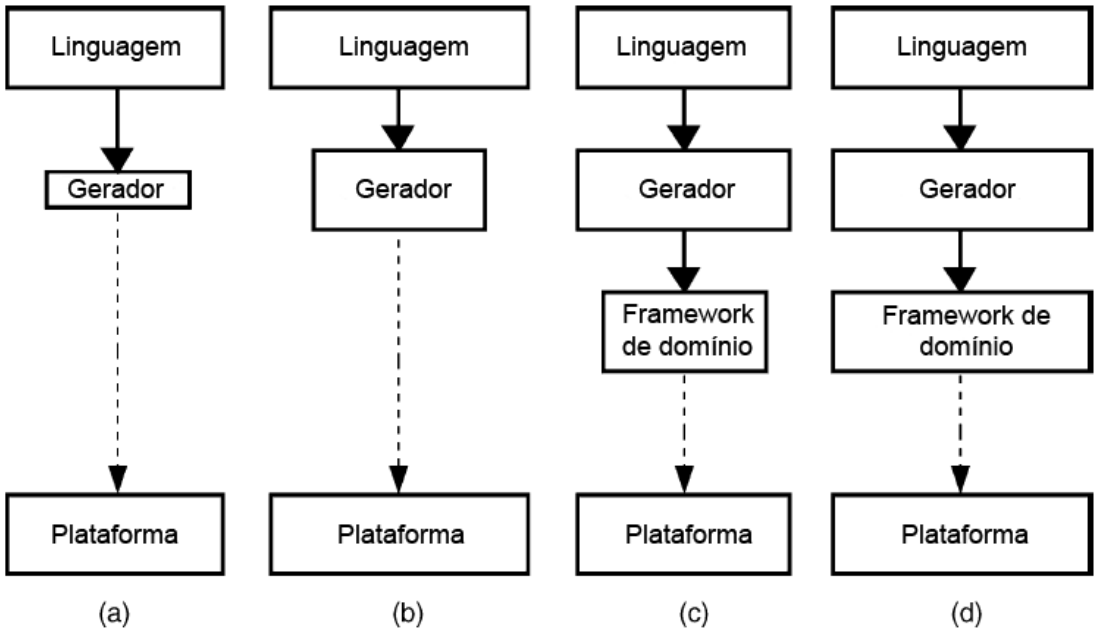


Figura 2.8: Algumas formas de dividir a abstração entre os componentes da DSM. Adaptado de (KELLY; TOLVANEN, 2008b)

são adequadas. Não é recomendado que geradores se encarreguem da maior parte do trabalho porque eles tendem a crescer com o tempo, dificultando sua manutenção. Além disso, deve-se certificar de que há um ganho significativo no nível de abstração com a utilização da DSL e de que ela contenha informações suficientes para que o gerador produza código completo. Diagramas de classes, por exemplo, não contém informações suficientes para que seja possível gerar a implementação dos métodos das classes e não elevam a abstração a um nível significativo, uma vez que possuem representações gráficas apenas para elementos de uma linguagem de programação orientada a objetos e não para conceitos de um domínio em particular, salvo diagramas que utilizam estereótipos (KELLY; TOLVANEN, 2008b).

2.3.3 Linguagem Específica de Domínio

Uma DSL é uma linguagem de modelagem específica de um domínio em particular. O que difere fundamentalmente uma DSL de uma linguagem de propósito geral (por exemplo, Java) é que a primeira é desenvolvida a partir do domínio do problema e não do domínio da solução. Idealmente, cada elemento da estrutura da DSL está diretamente relacionado a um conceito presente no domínio do problema e cada restrição desse domínio refere-se a uma ou mais restrições na linguagem. Grande parte dos conceitos é relacionada a "objetos" da linguagem, outros são mapeados como "propriedades" desses objetos, relacionamentos, submodelos e até mesmo referências a modelos feitos em outras linguagens. A aderência entre a DSL e o domínio do

problema proporciona muitos benefícios, como o aumento da produtividade, do nível de abstração e da qualidade do sistema. Essa aderência, no entanto, determina sua inutilidade em outros domínios (KELLY; TOLVANEN, 2008b).

Uma DSL, assim como qualquer linguagem, deve possuir sintaxe e semântica. A sintaxe define sua estrutura e a semântica define seu significado. A sintaxe de uma linguagem é, na verdade, dividida em duas: abstrata e concreta. A sintaxe abstrata é normalmente especificada em um metametamodelo e nela são definidas as construções da linguagem, suas propriedades e seus relacionamentos. Na sintaxe concreta, é definida a notação desses elementos, ou seja, a forma como eles são representados, que pode ser por meio de desenhos, tabelas, matrizes ou simples texto. A forma de representação escolhida deve estar em sincronia com os conceitos do domínio. Por exemplo, uma alavanca de um sistema de *infoentretenimento automotivo*[4] deve possuir uma ilustração similar na linguagem de modelagem. O princípio da fidelidade representacional (WEBER; ZHANG, 1996 apud KELLY; TOLVANEN, 2008b), que é definir apenas uma forma de representação para cada conceito do domínio, não apenas simplifica a definição da notação como também garante que todos os conceitos possam ser representados na linguagem.

A semântica de uma linguagem é a definição do significado das construções da linguagem. Como a linguagem é específica de um domínio em particular, a maioria de seus elementos carrega o significado proveniente do domínio. Por exemplo, no desenvolvimento de um software para celular, os conceitos *Câmera* e *Visor* possuem uma semântica bem definida. Já a semântica de uma linguagem de propósito geral não está relacionada a um domínio de negócio e, por isso, cabe aos desenvolvedores relacionar o domínio com a semântica da linguagem.

Para se projetar uma DSL, geralmente são utilizadas construções conhecidas de alguns modelos computacionais, como máquinas de estados, diagramas de fluxos de dados, entre outros. O que guia a escolha das construções a serem utilizadas é o domínio do problema. Alguns domínios possuem características predominantemente dinâmicas, levando ao uso, por exemplo, de máquinas de estados ou redes de Petri; outros são melhor representados em modelos estáticos e estruturais, como diagramas de *features* ou de classes. Na prática, ocorre a combinação desses modelos e as linguagens acabam cobrindo tanto a parte estrutural quanto a parte comportamental do domínio (KELLY; TOLVANEN, 2008b).

No entanto, uma única linguagem capaz de modelar diferentes características do domínio (por exemplo, estruturas de dados, concorrência e interação com o usuário) é grande demais, o que dificulta sua utilização e manutenção. Além disso, os modelos criados com essa linguagem

[4]Sistema embarcado em veículos que disponibiliza ao condutor diversas informações sobre o veículo e sobre estradas e mapas, além de tocar músicas, vídeos, entre outras funções.

são difíceis de modificar; dado um modelo contendo tanto a interface de usuário quanto a funcionalidade do sistema, qualquer alteração a ser feita na interface corre o risco de alterar também a funcionalidade, visto que ela encontra-se misturada com a modelagem da interface (KELLY; TOLVANEN, 2008b).

Uma solução para esse problema está na utilização de mais de uma linguagem de modelagem, cada uma modelando uma característica específica do domínio. Isso auxilia também no desenvolvimento do software, visto que partes do domínio são modeladas em diferentes fases desse processo (KELLY; TOLVANEN, 2008a).

2.3.4 Gerador de Código

O gerador é responsável por mapear o domínio do problema para o domínio da solução. Um dos geradores mais simples é aquele que gera uma parte de código fixo para cada elemento da DSL. O código gerado pode utilizar bibliotecas, componentes e *frameworks*, diminuindo assim a complexidade do gerador; por exemplo, um gerador que produz o código de um sistema para processar arquivos XML pode simplesmente acoplar esse código a um interpretador fornecido por terceiros ao invés de conter a lógica para construí-lo (KELLY; TOLVANEN, 2008b).

Para que o código gerado seja completo, ele deve conter tanto a parte estrutural quanto a comportamental, cuja automação é mais complexa. Além do código, os geradores também podem gerar casos de teste, simulações, protótipos, documentação de uso, scripts de configuração, medição, entre outros.

Uma característica fundamental para o sucesso da DSM é que o código gerado não precisa e nem deve ser modificado manualmente. Geradores são análogos a compiladores; não é necessário alterar o código de máquina gerado a partir de um programa em C. Toda alteração deve ser feita nos modelos.

No entanto, não há a necessidade de gerar o código completo da aplicação; pode-se utilizar frameworks de domínio e aproveitar os recursos da plataforma alvo. Por exemplo, um sistema distribuído pode utilizar EJBs (*Enterprise Java Beans*) como plataforma, retirando do gerador a responsabilidade de produzir uma estrutura completa de comunicação remota entre objetos.

O gerador pode ser usado para checar a validade e consistência dos modelos, uma vez que é impossível, em muitos casos, colocar todas as regras do domínio no metamodelo da linguagem – a ferramenta de modelagem, por tê-lo como base, não consegue garantir que nenhuma regra seja violada. Além disso, é comum a utilização de mais de uma linguagem de modelagem, resultando em vários modelos, possivelmente desenvolvidos por pessoas diferentes, cuja

integração precisa ser checada. O gerador também pode guiar a modelagem informando ao desenvolvedor as ações que devem ser tomadas.

O fato do código ser gerado e não mais escrito manualmente é que métricas baseadas em código para medir o esforço humano ou o tempo de desenvolvimento perdem sua validade; o código é apenas um subproduto dos modelos, os quais são considerados como sendo o próprio software. Visto que, na DSM, os desenvolvedores trabalham com os conceitos do domínio do problema ao invés do domínio da solução, as métricas devem emergir do próprio domínio, ou seja, deve-se utilizar os conceitos do domínio para calcular, por exemplo, o esforço humano necessário (KELLY; TOLVANEN, 2008b).

2.3.5 Framework de Domínio

Um framework de domínio é utilizado para conectar o código gerado com o ambiente de execução. Esse ambiente não é específico de um domínio e, por isso, o principal papel do framework é fornecer uma abstração desse ambiente do ponto de vista do domínio, ou seja, encapsular a melhor utilização possível do ambiente para o domínio da DSM. Ele não só define estruturas de dados, funções e componentes como também define um certo modelo de programação que o código gerado deve seguir. O código do framework, de modo geral, possui quatro propósitos: remover duplicidade do código gerado, fornecer templates para que o gerador preencha com código, integrar o código gerado com código existente e esconder a plataforma de execução (KELLY; TOLVANEN, 2008b).

Para remover o código duplicado, pode-se colocar no framework as estruturas de dados ou comportamentos que são comuns a todas as aplicações. Pode-se utilizar templates para definir a saída que o gerador produz; dessa forma, ao invés do gerador gerar todo o código, ele apenas preenche os templates pré-definidos. O framework também pode criar uma interface entre código gerado e código já existente. Por exemplo, ele pode definir alguns comportamentos básicos em classes abstratas; o gerador, então, produz subclasses que adicionam o comportamento modelado pelo desenvolvedor. Por fim, com o framework de domínio, é possível criar uma camada acima da plataforma de execução de tal forma que o código gerado fique independente de plataforma; por exemplo, um mesmo programa pode ser executado como um *applet* ou como um *midlet*[5]. O framework de domínio, no entanto, nem sempre é necessário. Existem várias plataformas de execução que já fornecem inúmeros recursos e que podem ser utilizados diretamente pelo código gerado (KELLY; TOLVANEN, 2008b).

[5]Classe Java ME (Micro Edition) utilizada para programar aplicativos para celulares.

2.3.6 DSL *versus* GPL (*General Purpose Language* – Linguagem de Propósito Geral)

Segundo Mernik, Heering e Sloane (2005, p. 317-320), as vantagens de se utilizar DSL em relação às GPLs são:

- As construções oferecidas são geralmente mais apropriadas ao domínio do que as construções de uma GPL, mesmo quando são utilizados recursos da GPL que permitem a definição personalizada de construções (por exemplo, macros e templates em C++);
- Nem todos os conceitos de um domínio podem ser mapeados em objetos ou funções em uma GPL. Um exemplo típico são alguns interesses transversais, cujo encapsulamento é difícil de alcançar, mesmo utilizando linguagens orientadas a aspectos;
- O uso de DSLs oferece possibilidades de análise, verificação, otimização, paralelização e transformação em termos de construções de DSLs que seriam muito mais difíceis ou inviáveis se uma GPL fosse usada porque os padrões de código envolvidos são muito complexos ou não são bem definidos;
- DSLs permitem o reuso de vários artefatos de software, como abstrações do domínio, arquiteturas de software, além do próprio código fonte.

Entretanto, o desenvolvimento de DSLs exige conhecimento não só do domínio do problema como também de técnicas de desenvolvimento de linguagens, o que poucos desenvolvedores possuem. Por causa disso, há o custo de treinamento dessas técnicas, que é somado ao custo de desenvolvimento e manutenção de DSLs. Além disso, deve-se averiguar se o desenvolvimento de uma DSL em um determinado domínio valerá a pena, pois é difícil definir o escopo apropriado de uma DSL (DEURSEN; KLINT; VISSER, 2000). Os conceitos básicos de uma DSL adequada podem emergir depois de ter sido feita muita programação em GPLs (MERNIK; HEERING; SLOANE, 2005).

Capítulo 3

TRABALHOS RELACIONADOS

Neste capítulo são apresentados sucintamente alguns trabalhos relacionados encontrados na literatura. Na Seção 3.1 são abordados alguns trabalhos sobre construção de DSLs e na Seção 3.2 são apresentadas algumas DSLs para sistemas embarcados.

3.1 Construção de DSLs

Mernik, Heering e Sloane (2005) descrevem as diferentes fases de desenvolvimento de uma DSL e várias diretrizes, chamadas pelos autores de *padrões*, que podem ser utilizadas em cada uma delas. As fases são: (a) Decisão; (b) Análise; (c) Projeto; (d) Implementação; e (e) Implantação. Essas diretrizes visam responder *quando* e *como* desenvolver DSLs, sendo que a fase de Decisão está relacionada ao *quando* e as demais fases, ao *como*. As diretrizes de diferentes fases são independentes, i.e., pode haver qualquer combinação de diretrizes de diferentes fases.

Na fase de Decisão, tem-se nove diretrizes, que estão descritas abaixo:

Notação Deve-se considerar o desenvolvimento de uma DSL quando houver a necessidade de notações (ou construções) mais apropriadas para representar o domínio. Pode-se considerar seu desenvolvimento também quando houver a necessidade de transformar uma notação visual para textual ou quando precisa-se adicionar notações amigáveis a uma API existente.

AVOPT A Análise, Verificação, Otimização, Paralelização e Transformação de sistemas geralmente não são viáveis quando os mesmos são escritos com GPLs porque os padrões de código são muito complexos ou não são bem definidos. Uma DSL adequada permite a realização mais facilitada dessas atividades.

Automação de Tarefas É comum programadores gastarem tempo com tarefas de programação repetitivas e que geralmente servem apenas para satisfazer as regras da GPL e não do domínio (por exemplo, métodos de uma interface Java que devem ser implementados mesmo quando não utilizados). Nesses casos, pode-se criar uma DSL apropriada tal que seu compilador gere automaticamente o código necessário.

Linha de Produtos Os membros de uma linha de produtos compartilham uma arquitetura comum e são desenvolvidos com elementos básicos de um conjunto comum. O uso de uma DSL pode facilitar a especificação desses elementos.

Representação de Estruturas de Dados Muitos softwares são orientados a dados e possuem estruturas de dados grandes e complexas. Pode-se, então, utilizar uma DSL para representá-las de forma mais simples.

Travessia de Estruturas de Dados Pode-se utilizar uma DSL para expressar mais claramente travessias em estruturas de dados complexas.

Front-End de Sistemas Uma DSL pode ser utilizada para configurar e adaptar sistemas.

Interação Sistemas em que o usuário interage com menus ou mesmo texto para obter o resultado desejado pode possuir uma DSL que permite a especificação de entradas complicadas e/ou repetitivas. Exemplos são sistemas que fornecem uma linguagem para a especificação de macros, como OpenOffice[1] e jEdit[2].

Construção de Interfaces de Usuário Geralmente é feita utilizando DSLs.

Na fase de Análise, deve-se identificar e conhecer o domínio do problema e isso pode ser feito por meio de documentos técnicos, *experts* do domínio, códigos existentes, e usuários finais. O resultado da análise consiste basicamente de uma terminologia do domínio e sua semântica. Mernik, Heering e Sloane (2005) identificaram três diretrizes para realizar essa análise:*Informal*, em que a análise é feita de forma informal; *Formal*, utilizando algum método de análise de domínio, como DARE (*Domain Analysis and Reuse Environment* – Análise de Domínio e Ambiente de Reuso) (FRAKES; PRIETO-;DIAZ; FOX, 1998) ou ODE (*Ontology-based Domain Engineering* – Engenharia de Domínio baseada em Ontologia) (FALBO; GUIZZARDI; DUARTE, 2002); e *Inspeção de Código* (*Extract from code*), em que o conhecimento sobre o domínio é extraído de código-fonte existente, de forma manual ou por meio de ferramentas apropriadas.

[1] http://www.openoffice.org/
[2] http://www.jedit.org

Na fase de Projeto, há duas diretrizes, as quais são *Exploração de Linguagens* e *Invenção de Linguagens*. Na primeira, são aproveitadas partes de GPLs ou até mesmo DSLs existentes e há três subdiretrizes a essa: *Piggyback*, *Especialização* e *Extensão*. Na *Piggyback*, uma linguagem é parcialmente utilizada e, na *Especialização*, uma linguagem é restringida. A diferença entre elas está em quanto a barreira entre a DSL e o restante da linguagem é rígida.

Na diretriz *Invenção de Linguagens*, as DSLs não possuem relação com nenhuma linguagem existente e, na prática, seu desenvolvimento é mais difícil. Os critérios aplicáveis às GPLs, como legibilidade, simplicidade e ortogonalidade, tem sua validade para DSLs mas o desenvolvedor também deve levar em consideração o caráter especial das DSLs bem como o fato de que, às vezes, seus usuários não precisam ser programadores.

Na fase de Implementação, há sete diretrizes:

Interpretador As construções da DSL são interpretadas usando o ciclo *obtém, decodifica e executa instrução*. Essa diretriz é apropriada para linguagens com características dinâmicas ou cuja velocidade de execução não importa. Comparada à compilação, a interpretação é mais simples, proporciona maior controle da execução e é fácil de ser estendida.

Compilador/Gerador de Aplicações As construções da DSL são traduzidas para construções de uma linguagem base e chamadas a bibliotecas. Nessa diretriz, é possível fazer uma análise estática completa do sistema.

Preprocessador As construções da DSL são traduzidas para construções em uma linguagem existente, porém, a análise estática é limitada a essa linguagem. Há quatro subdiretrizes:

Processamento de macros Expansão de definições de macros.

Transformação Source-to-source O código fonte escrito com a DSL é transformado em outra linguagem.

Pipeline Há um encadeamento de (pre)processadores, cada um manipulando uma sublinguagem da DSL, em que a saída de um é a entrada de outro.

Processamento Léxico Requer apenas um simples exame léxico, sem a necessidade de analisar árvores sintáticas. Um exemplo é a linguagem SSC, que é utilizada para a composição de subsistemas de software (BUFFENBARGER; GRUELL, 2001).

Embutimento As construções da DSL são embutidas em uma GPL existente, a qual é chamada de *linguagem host*, na forma de novos tipos abstratos de dados e operadores, por exemplo. Utiliza-se dos recursos de criação de construções definidas pelo usuário oferecidos pela linguagem, por exemplo, pode-se utilizar *templates* em C++.

Compiladores/Interpretadores Extensíveis Um compilador/interpretador é estendido com regras de otimização e/ou geração de código específico de domínio. Geralmente, compiladores são mais difíceis de estender que interpretadores.

Produtos Comerciais de Prateleira (*Commercial Off-The-Shelf*) Produtos comerciais existentes podem ser utilizados de forma restritiva a um domínio particular. Por exemplo, pode-se utilizar o Powerpoint somente para fazer digramas de fluxos de dados; pode-se utilizar XML para desenvolver DSLs, em que a gramática da linguagem é escrita utilizando DTD[3] e que os arquivos em XML podem ser lidos e transformados com *parsers* existentes.

Híbrido Uma combinação das diretrizes anteriores.

Evermann e Wand (2005) propuseram um método para restringir a sintaxe de uma linguagem de modelagem de propósito geral por meio de uma ontologia de conceitos do domínio de tal forma que a linguagem resultante ofereça apenas possibilidades válidas de modelagem a esse domínio, tornando-a uma linguagem de modelagem específica de domínio. O método possui quatro passos:

- Atribuição das semânticas do domínio às construções da linguagem (mapeamento entre a ontologia e a linguagem). Nesse passo, duas perguntas devem ser respondidas:
  - Como um elemento do domínio (conceito ontológico) pode ser representado na linguagem escolhida?
  - Como uma construção da linguagem pode ser interpretada em termos de um domínio (ontologicamente)?

  Para responder a essas perguntas, são criados mapeamentos de conceitos da ontologia para construções da linguagem e vice versa que, juntos, atribuem semânticas ontológicas à linguagem. Na ontologia definida por Bunge (1983), por exemplo, o mundo é feito de *coisas* (por exemplo, carro e pessoa) que possuem *propriedades gerais* (pertencentes a um conjunto, por exemplo, cor e nome) e também *propriedades individuais* (pertencentes a uma coisa em particular, por exemplo, carro de cor azul e pessoa de nome Adriana). O *estado* de uma *coisa* é, basicamente, a configuração de todas as suas *propriedades individuais* no tempo. Um possível mapeamento entre esses conceitos e a UML é mostrado na Tabela 3.1.

[3] *Data Type Definition* – arquivo que basicamente define a estrutura de um XML.

Tabela 3.1: Mapeamento entre alguns conceitos da ontologia Bunge e a UML

Conceito ontológico	Construção da UML
Coisa	Objeto
Propriedade Geral	Atributo
Propriedade Individual	Link de Atributo
Estado	Estado (do diagrama de estados)

- Identificação das suposições (regras) que afetam os elementos da ontologia e seus relacionamentos. Por exemplo, na ontologia supracitada, uma *mudança* é a alteração de uma *propriedade individual* de uma determinada *coisa*. Uma das suposições feitas por Bunge (1983) que envolve esse conceito foi: cada *mudança* é uma mudança de *estado* em um determinado modelo;

- Transferência dessas pressuposições por meio do mapeamento feito no passo 1, resultando em regras que restringem o uso das construções da linguagem e limitam os tipos de afirmações que podem ser feitas sobre o domínio. Nas regras, os termos utilizados para se referirem aos conceitos do domínio são as construções da linguagem mapeada. Um exemplo de regra obtida, tendo como base a ontologia mencionada e a UML, é: Cada *estado (do diagrama de estados)* é definido em termos dos *valores dos atributos* (*propriedades individuais*); e

- Modificação da linguagem para atender a essas regras. As modificações necessárias para a implementação da regra supramencionada estão na Figura 3.1, parte (b). O restante do metamodelo foi omitido em ambas as partes.

A linguagem específica de domínio resultante, no entanto, serve apenas para criar modelos conceituais e, portanto, não visa a geração automática de código.

Günther, Haupt e Splieth (2010) apresentam um processo ágil e leve para a construção de DSLs embarcadas em linguagens dinâmicas. O processo é dividido em três fases: *Domain Design* (Projeto do Domínio), *Language Design* (Projeto da Linguagem) e *Language Implementation* (Implementação da Linguagem). Os autores também apresentam vários padrões de construção de DSLs que podem ser usados na última fase, os quais são fortemente relacionados aos seguintes conceitos: *Modelagem da Linguagem* (quais construções da linguagem implementam os conceitos do domínio), *Integração da Linguagem* (como facilmente integrar a DSL com outros componentes) e *Purificação da Linguagem* (como otimizar características da linguagem, como por exemplo, legibilidade). Os autores afirmam que "o uso desse processo no contexto

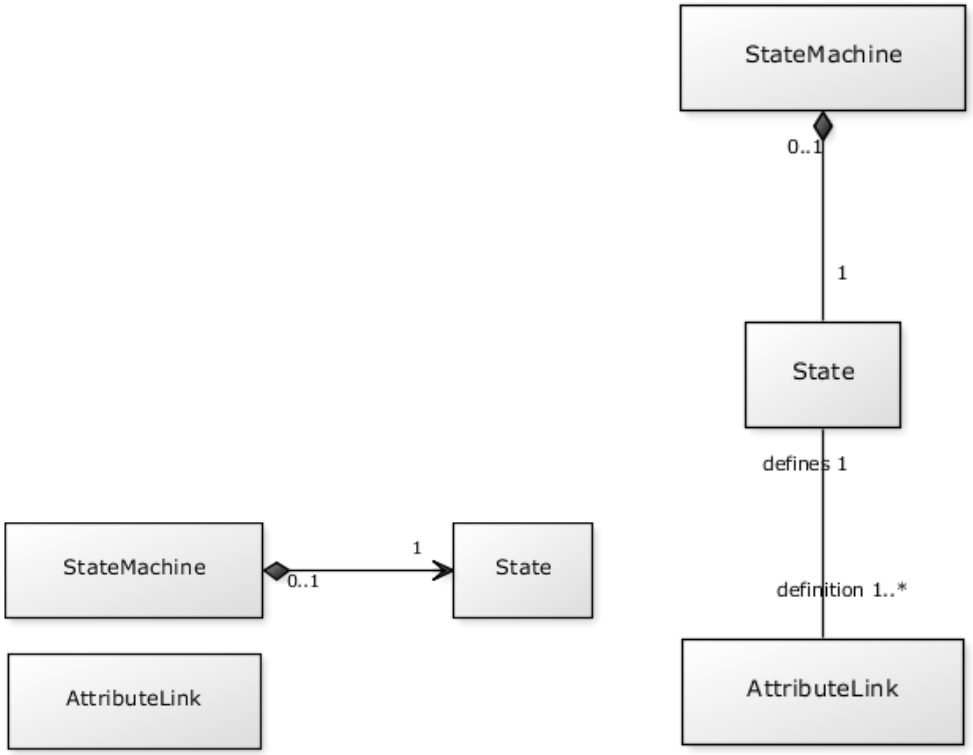


Figura 3.1: Mudanças do metamodelo propostas. (a) Metamodelo original (b) Metamodelo modificado

de desenvolvimento de aplicações permite que as DSLs sejam subprodutos (desse desenvolvimento) que aumentam de forma imediata a produtividade para o projeto atual e (também) são passíveis de uso futuro".

Robert et al. (2009) definem um processo leve para a construção de perfis UML; em outras palavras, criar linguagens específicas de domínio pela especialização de uma linguagem de propósito geral (que, neste caso, é a UML). O processo possui três atividades: *Problem Description* (Descrição do Problema), *Refinement and Restrictions* (Refinamento e Restrições), e *Profile Definition* (Definição do Perfil). Os artefatos resultantes são respectivamente um *Modelo do Problema*, um *Modelo do Domínio* (ambos expressos com diagramas UML) e um *Perfil*. Há dois atores: o *expert do domínio* e o *expert de linguagens*. O primeiro é responsável pela primeira atividade, o último é responsável pela última e ambos trabalham na segunda atividade. O processo proposto também emprega um conjunto de heurísticas predefinidas que geram de forma automática um perfil incompleto a partir do Modelo do Domínio. Este perfil é então otimizado com várias diretrizes definidas pelos autores. O intuito dessas diretrizes é assegurar a correção e otimizar os perfis.

Strembeck e Zdun (2009) propõem um processo sistemático de desenvolvimento de DSLs,

o qual teve como base a experiência que obtiveram com diferentes projetos, protótipos e experimentos de DSLs. O processo possui quatro atividades principais: *Defining the DSL's core language model* (Definir o modelo do núcleo da linguagem), *Defining the behavior of DSL language elements* (Definir o comportamento dos elementos da linguagem), Defining the DSL's concrete syntax(es) (Definir a(s) sintaxe(s) concreta(s) da DSL) e *Integrating DSL artifacts with the platform/infrastructure* (Integrar os artefatos da DSL com a plataforma/infraestrutura). Essas atividades produzem os seguintes artefatos, respectivamente: o modelo do núcleo da linguagem, a definição de comportamentos, as sintaxes concretas e as regras de transformação. Os primeiros três artefatos compõem a DSL enquanto que o último está relacionado à plataforma de desenvolvimento. Essas atividades possuem subatividades e todas elas possuem fluxos de controle bem definidos, embora não rígidos. Os autores também apresentam atividades para "adaptar o processo à abordagem padrão de desenvolvimento da empresa correspondente", relacionando o tipo do projeto, as pessoas envolvidas, o orçamento, entre outros fatores. Como essa abordagem expõe as tarefas específicas de desenvolvimento de DSLs, ela garante a consideração explícita delas no processo de desenvolvimento.

3.2 DSLs para Sistemas Embarcados

Existem diversas DSLs na literatura direcionadas a sistemas embarcados e a robôs móveis (ROPER; OLSSON, 2005; BALASUBRAMANIAN et al., 2007; HAMMOND; MICHAELSON, 2003a; KONOLIGE, 1997; FIRBY, 1994; BROOKS, 1990; PETERSON; HAGER, 1999). Há a DSL textual CATAPULTS (ROPER; OLSSON, 2005) para programar escalonadores de *threads* para sistemas embarcados. Como os sistemas embarcados frequentemente precisam executar diversas tarefas concorrentemente (por exemplo, controlar diferentes componentes de *hardware*), o algoritmo de escalonamento de *threads* tem um impacto significativo no seu desempenho. Essa DSL tenta resolver esse problema permitindo que o desenvolvedor crie escalonadores específicos e otimizados para uma determinada aplicação. Isso também faz com que os escalonadores fiquem modulares, uma vez que ficam separados das aplicações, e previne erros que são frequentes na programação de baixo nível em C ou Assembly.

Considere uma estação de monitoramento de clima. É uma aplicação que precisa monitorar vários sensores de temperatura em frequências diferentes, controlar um visor que muda quando a temperatura atinge um certo limiar e executar cálculos diversos quando o hardware estiver inativo. Por ser uma aplicação multitarefa, pode-se criar uma *thread* para cada sensor de temperatura, uma *thread* para controlar o visor e uma ou mais *threads* para executar os cálculos diversos. Porém, as *threads* dos sensores lentos não devem ser acionadas tão frequentemente

quanto as *threads* dos sensores rápidos porque aqueles sensores nem sempre estarão prontos para dar suas informações. Com a CATAPULTS, é possível criar um escalonador específico para essa aplicação e que seja mais eficiente do que os algoritmos de escalonamento convencionais; isso porque é possível especificar, por exemplo, que a *thread* do visor só será executada quando a temperatura atingir 100ºF. Nas Listagens 3.1 e 3.2 tem-se o código do escalonador para essa aplicação, o qual foi dividido em duas partes e pequenos detalhes omitidos por questões de espaço.

Na Listagem 3.1, linhas 2 a 4, é feita a definição de quais informações sobre cada *thread* o escalonador deve saber; nesse caso, apenas o seu estado, que pode ser *nova*, *em execução*, *suspensa*, etc. Nas linhas 6 a 8, há a definição de variáveis provenientes da aplicação que devem ser importadas ao escalonador por *thread*. Isso permite ao escalonador monitorar mudanças feitas a essas variáveis e também alterá-las, tornando-se uma maneira de se comunicar com a aplicação. No caso da aplicação, a variável a ser importada da aplicação é a *threadclass*, que indica o tipo de *thread* (se é de sensor rápido, sensor lento, de visor ou de cálculos). Da linha 10 à 26 são definidas diversas variáveis globais internas a serem usadas pelo escalonador. A última parte da Listagem 3.1, a construção *imports*, determina variáveis de aplicação globais a serem importadas para o escalonador. Nesse caso, apenas a variável *temperature* é importada, a qual indica a temperatura total calculada pela aplicação.

Na Listagem 3.2, há a parte de definição de *manipuladores de eventos* (*event handlers*) do escalonador. Os eventos são disparados sempre que é necessário realizar alguma ação de escalonamento. Alguns eventos são: (a) *init*, chamado na inicialização do escalonador; (b) *newthread*, chamado quando uma nova *thread* é criada pela aplicação; e (c) *schedule*, chamado para escalonar as *threads*.

No manipulador do evento *init* na Listagem 3.2, são inicializadas as variáveis que guardam a quantidade de escalonamentos feitos desde a última vez em que um determinado tipo de *thread* (de visor, de sensor lento ou de sensor rápido) fora executado. Por exemplo, se o valor de *last_sensor1* for 3 significa que as últimas três *threads* escalonadas não foram de sensores rápidos. No evento *newthread*, o escalonador simplesmente coloca a *thread* recém-criada na pilha de novas *threads*, definida na estrutura *data* apresentada na Listagem 3.1, linhas 10-26. O manipulador do evento *schedule* é o algoritmo de escalonamento propriamente dito. Da linha 14 à 23, há a alocação das novas *thread*s em suas respectivas filas; se a *thread* for de sensor rápido (*SENSOR1CLASS*), ela será adicionada na fila de sensores rápidos (*standard_sensors*), e assim por diante. O restante do algoritmo se encarrega de despachar a *thread* apropriada para execução, de acordo com os requisitos. Na linha 27, por exemplo, o algoritmo verifica se a

Listagem 3.1: Exemplo de escalonador para estação de monitoramento de clima: parte da definição de dados

```
scheduler station {
  thread {
    int state; // nova, em execução, suspensa, etc.
  }

  threadimports {
    int threadclass default 0; // se é de sensor, do visor, ou de cálculos
  }

  data {
    threadref current; // thread atual
    threadref next; // próxima thread
    stack NQ; // novas threads
    queue standard_sensors; // sensores
    queue slow_sensors; // sensores que serão monitorados
                        // menos frequentemente
    threadref display; // thread do visor
    queue calculations; // thread para os cálculos diversos
    // última vez em que as threads foram executadas
    // sensor1 são os sensores normais e sensor2, os lentos
    int last_display, last_sensor1, last_sensor2;

    const int UNKNOWNCLASS = 0,
              SENSOR1CLASS = 1, SENSOR2CLASS = 2,
              DISPLAYCLASS = 3, CALCCLASS = 4;
}

  imports {
    int temperature default 0;
  }

  // definição de eventos omitida
}
```

temperatura total ultrapassou 100º e se faz mais de dez vezes que a *thread* do visor não foi escalonada; em caso afirmativo, essa *thread* é despachada para execução.

Hammond e Michaelson (2003b) apresentam uma DSL textual, funcional e concorrente para sistemas embarcados de tempo real chamada *Hume*. Os autores identificaram cinco propriedades essenciais ou desejadas em uma linguagem para esses sistemas, as quais são:

Determinabilidade a linguagem deve permitir a construção de sistemas determinísticos, i.e., em condições idênticas, todas as execuções do sistema devem ser equivalentes;

Tempo/espaço limitado a linguagem deve permitir a construção de sistemas cujos recursos são estaticamente limitados, por exemplo, dado um tamanho fixo de memória, a linguagem não deve permitir a especificação de sistemas que ultrapassem essa capacidade;

assincronia a linguagem deve permitir a construção de sistemas que são capazes de responder a entradas assim que elas são recebidas, sem impor uma ordem nas interações;

concorrência a linguagem deve permitir a construção de sistemas como unidades que compu-

tam independentemente e que se comunicam entre si;

exatidão a linguagem deve permitir um alto grau de confiança em que os sistemas cumprirão seus requisitos formais.

Listagem 3.2: Escalonador para estação de monitoramento de clima: definição de eventos

```
scheduler station {
  // definiçao dedados omitida
  event init {
    last_display = 0;
    last_sensor1 = 0;
    last_sensor2 = 0;
  }
  event newthread(t) {
    t => NQ; // coloca t na pilha de novas threads NQ
  }
  event schedule {
    threadref tmp;
    // moveras threads novas para suas respectivas filas
    foreach tmp in NQ {
      if (tmp.threadclass == SENSOR1CLASS)
        tmp => standard_sensors;
      else if (tmp.threadclass == SENSOR2CLASS)
        tmp => slow_sensors;
      else if (tmp.threadclass == DISPLAYCLASS)
        tmp => display;
      else if (tmp.threadclass == CALCCLASS)
        tmp => calculations;
    }
         last_display++; last_sensor1++; last_sensor2++;
                if (|next| == 1) { // |next| = tamanho de 'next'
                        next => current;
                } else if (temperature>=100 && last_display>10) {
                        display => current;
                        last_display = 0;
                } else if (last_sensor1>3 && |standard_sensors|>0) {
                        standard_sensors => current;
                        last_sensor1 = 0;
                } else if (last_sensor2>6 && |slow_sensors|>0) {
                        slow_sensors => current;
                        last_sensor2 = 0;
                } else {
                        calculations => current;
                }
                dispatch current;
        }
}
```

O objetivo da linguagem *Hume* é aumentar o nível de abstração dos sistemas de tempo real mantendo essas propriedades. Dentre os recursos da linguagem pode-se citar tratamento de exceções, gerenciamento automático de memória, funções de ordem superior[4], polimorfismo e recursão. A linguagem também fornece o recurso de metaprogramação, que permite a construção de *templates* e macros para, por exemplo, encapsular definições repetitivas, entre outras.

As duas unidades básicas da linguagem são *box* e *wire*. *Box* é uma abstração de uma máquina de Moore, que é basicamente uma máquina de estados finita que produz uma saída

[4]Funções cuja entrada pode receber funções e cuja saída pode ser uma função.

com base em uma entrada. Apesar do corpo de uma *box* ser uma única função, o processo definido pela *box* é iterado indefinidamente, executando a função várias vezes. Como uma *box* não guarda estado (*stateless*), as informações que devem ser preservadas entre as iterações devem ser declaradas explicitamente por meio de um *wire*. Um *wire* basicamente conecta uma saída a uma entrada. Tal saída/entrada pode ser uma porta, um fluxo de dados ou vir de uma *box*.

Na Listagem 3.3 é mostrado um programa que verifica e exibe a paridade de um fluxo de dados. Na primeira linha há simplesmente a definição de dois tipos de dados, *bit* (apenas um caractere) e *parity* (um valor booleano). Em seguida, há a definição da *box even_parity* (paridade par), cuja entrada (linha 4) é uma dupla de um *bit* e uma *parity* e a saída (linha 5) é uma dupla de uma *parity* e uma *string*. Nas linhas 7-10 há as possíveis combinações de valores entre essas duplas. A linha 7, por exemplo, determina que quando *b* for 0 e *parity* for *true*, *p'* será *true* e *show* será *"true"*. O comando *unfair* determina basicamente que as combinações sejam feitas sequencialmente; a explicação completa desse recurso, no entanto, está além do escopo deste trabalho. Da linha 12 em diante, há a criação de duas variáveis,*input* e *output*, representando os fluxos de dados de um sensor qualquer e da saída padrão, respectivamente, e a definição de vários *wires*. O primeiro conecta o fluxo do sensor à variável *b* da box *even_parity*, o segundo conecta as variáveis *p* e *p'* com o objetivo de preservar suas informações entre as *iterações* da *box* e o terceiro conecta a variável *show* à saída padrão, onde será mostrado o resultado do programa.

Listagem 3.3: Exemplo de verificador de paridade com *Hume*

```
1  type bit = word 1; type parity = boolean;
2
3  box even_parity
4  in ( b :: bit, p :: parity )
5  out (p' :: parity, show :: string)
6  unfair
7    ( 0, true  ) -> ( true,  "true"  )
8  | ( 1, true  ) -> ( false, "false" )
9  | ( 0, false ) -> ( false, "false" )
10 | ( 1, false ) -> ( true,  "false" );
11
12 stream input from "/dev/sensor";
13 stream output to "std_out";
14
15 wire input to even_parity.b;
16 wire even_parity.p' to even_parity.p initially true;
17 wire even_parity.show to output;
```

Durelli et al. (2010) apresentam uma DSL gráfica para o desenvolvimento de robôs móveis autônomos ou teleoperados, embasada nos conceitos de linha de produtos de software, particularmente no conceito de diagrama de *features*. Os autores apresentam um processo ágil para extração das *features* comuns e variáveis do domínio pela análise de três ou mais aplicações

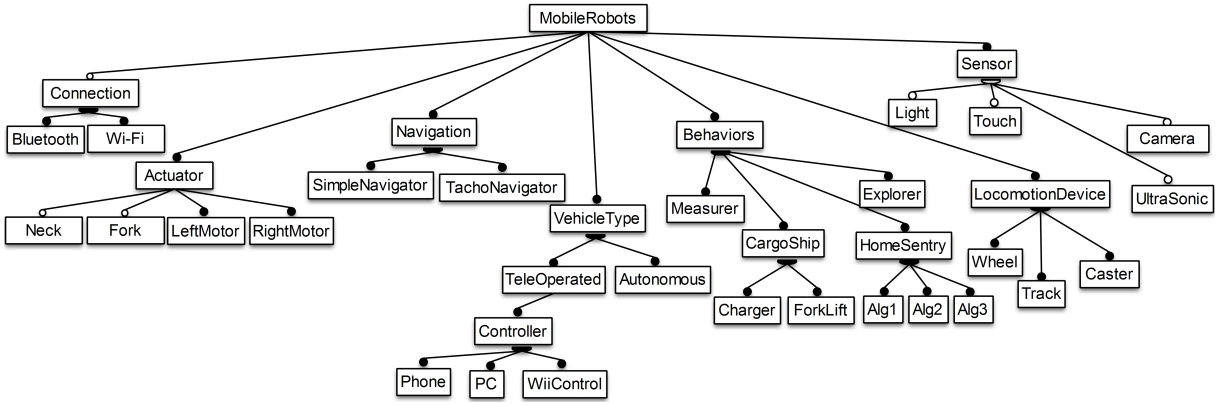


Figura 3.2: Modelo de Features(DURELLI et al., 2010)

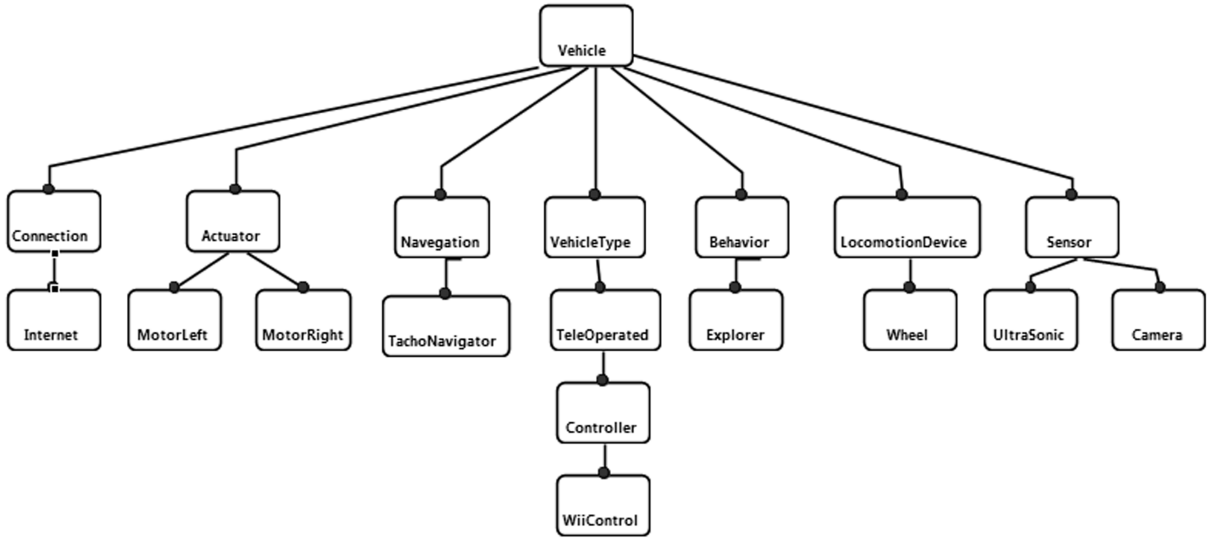


Figura 3.3: Exemplo de um sistema de controle de robô móvel(DURELLI et al., 2010)

diferentes, pelo qual a DSL fora criada. O desenvolvimento da aplicação se dá pela seleção das *features* disponíveis, de acordo com seus tipos: se obrigatória, alternativa ou opcional. No entanto, não é possível especificar novos comportamentos para o robô utilizando diretamente a DSL porque os que estão disponíveis são somente os provenientes das aplicações que foram utilizadas para criá-la. Os novos comportamentos devem ser implementados manualmente e o processo deve ser refeito com o intuito de agregá-los à DSL.

Na Figura 3.2 é apresentado o modelo de *features*. Nesse modelo, estão representadas todas as *features* que podem ser utilizadas para criar aplicações.

Na Figura 3.3 é apresentada uma aplicação feita com a DSL. Nessa aplicação, o robô utiliza dois sensores dos quatro disponíveis, *UltraSonic* e *Camera*, e é teleoperado com um controle do Nintendo Wii, representado pela *feature WiiControl*.

Trabalhos	ROBERT et al	GUNTHER et al	STREMBECK e ZDUN	Abordagem atual
Características				
Metamodelo Genérico	-	-	-	X
Análise da parte dinâmica do domínio	-	X	X	X
Recursos usados na análise do domínio	Experts	Experts/Documentos	Não definido	Experts/Documentos/Hardware
Geração automática de metamodelo	X	-	-	-
Padrões de projeto de metamodelos	X	X	-	-
Tipo de linguagem	Interna/Gráfica	Interna/Textual	Independente	Externa/Gráfica
Dirigido a um domínio especifico	-	-	-	X

Tabela 3.2: Comparativo entre os trabalhos relacionados e a abordagem proposta

3.3 Considerações Finais

Os trabalhos diretamente relacionados estão descritos na Seção 3.1. Na Tabela 3.2 tem-se um comparativo entre a abordagem proposta e os trabalhos dos autores Robert et al. (2009), Günther, Haupt e Splieth (2010) e Strembeck e Zdun (2009). Nessa tabela, há na primeira coluna uma relação de algumas características presentes nos modelos de processo de criação de DSLs, as demais colunas representam os trabalhos e cada célula relaciona uma característica a um trabalho e informa se essa está presente no mesmo.

O desenvolvimento de um metamodelo genérico que suporte a criação de DSLs está presente apenas na abordagem apresentada por este trabalho. Durante a análise do domínio, em praticamente todos os trabalhos são levados em consideração o conhecimento dos experts juntamente com documentos, modelos, código-fonte, etc. Na abordagem apresentada, além desses itens é analisado também os componentes físicos, o hardware e como eles funcionam, e dessas informações extraem-se abstrações do domínio. O tipo de linguagem desenvolvida neste trabalho também difere dos demais; nenhum deles trabalham diretamente com linguagens externas e gráficas. Além disso, somente a abordagem apresentada é dirigida especificamente a um domínio; nesse caso, a robótica móvel.

O trabalho de Robert et al. (2009) é o único que apresenta a modelagem do domínio utilizando apenas diagramas de classes, que descrevem somente as partes estáticas do domínio. Todos os demais levam em consideração suas características dinâmicas.

Por outro lado, há duas características importantes que este trabalho não aborda e que estão presentes em Robert et al. (2009) e Günther, Haupt e Splieth (2010), as quais são a definição de padrões de projeto para o desenvolvimento de linguagens e de heurísticas que permitem a criação automática do metamodelo, ainda que incompleto (presente somente no primeiro trabalho). Isso potencializa o aumento da produtividade e também da qualidade das linguagens criadas.

No entanto, um objetivo comum entre todos os trabalhos é o fornecimento de um processo leve porém suficiente para produzir pelo menos uma versão inicial de uma DSL que seja capaz de melhorar a produtividade do desenvolvimento.

Capítulo 4

ABORDAGEM PARA CRIAÇÃO DE LINGUAGENS ESPECÍFICAS DE DOMÍNIO PARA ROBÓTICA MÓVEL

O processo de desenvolvimento de DSLs explorado foi estruturado em cinco atividades. Essas atividades são comuns Na Figura 4.1 tem-se um diagrama SADT ilustrando o processo. Cada atividade produz um artefato que é utilizado para a confecção da DSL; algumas delas dependem de itens de controle (setas que incidem no topo) e de certos mecanismos para a sua execução (setas que incidem na base). O responsável por conduzir essas atividades é chamado de engenheiro de domínio. A atividade de análise está relacionada à atividade "Identificar declarações do domínio" e a atividade projeto, à atividade "Criar o modelo da linguagem". As cinco atividades são explicadas nas próximas seções juntamente com sua aplicação no desenvolvimento de uma DSL, aqui chamada de DSL 1.

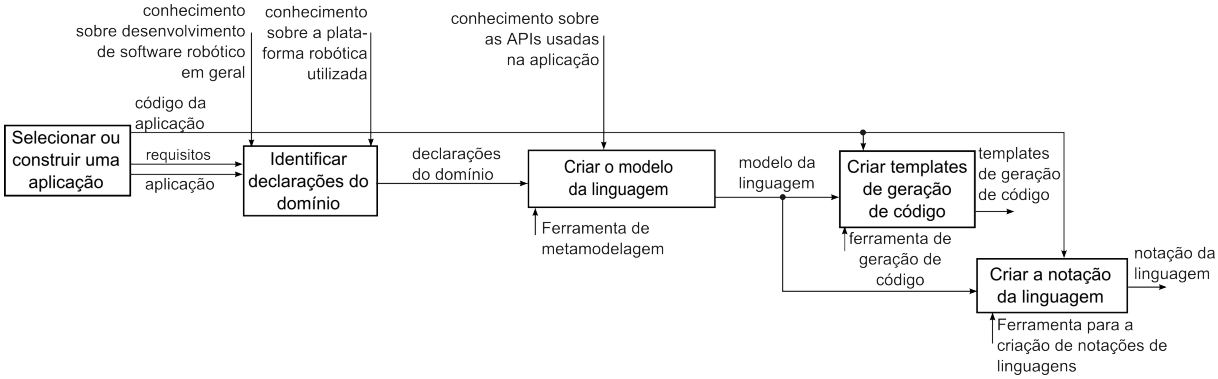


Figura 4.1: Diagrama SADT do Processo de Desenvolvimento de DSLs

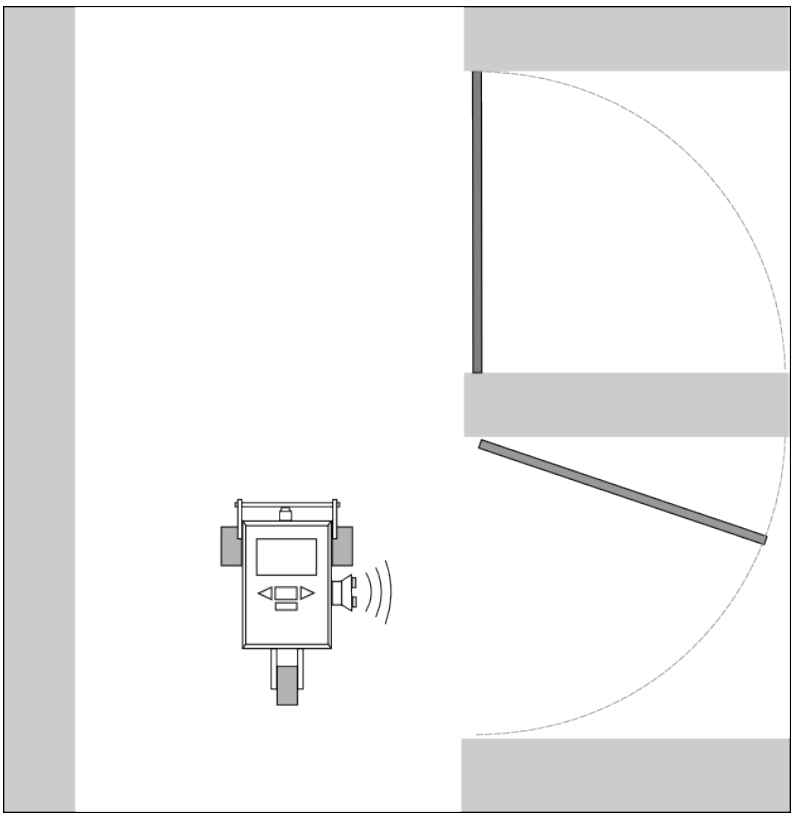


Figura 4.2: Ilustração do robô identificador de portas abertas e seu ambiente

4.1 Atividade "Selecionar ou construir uma aplicação"

Nesta atividade, uma aplicação de robôs móveis é selecionada pelo engenheiro de domínio ou escrita pelo desenvolvedor. É necessário que o engenheiro de domínio tenha conhecimento dos requisitos dessa aplicação e também das bibliotecas e *frameworks* usados para implementá-la, pois tal conhecimento é utilizado na próxima atividade e também em outras atividades do processo.

Como exemplo, foi construído um robô móvel cujo objetivo é andar por um corredor e soltar um alarme sonoro sempre que encontrar uma porta aberta, dando meia-volta quando terminar o corredor. O código-fonte pode ser visto no Apêndice A. Para isso foi utilizado o kit LEGO Mindstorms, composto de um microcontrolador, três motores, diferentes sensores e uma coleção de peças tradicionais da LEGO.

Na Figura 4.2 está ilustrada a configuração do robô, que possui direção diferencial e um sensor ultrassônico na lateral cujo objetivo é medir a distância entre o mesmo e as paredes e portas. Há também um sensor de toque na parte frontal para identificar o final do corredor. Ainda na mesma figura há a ilustração de um corredor com duas portas, uma aberta e outra fechada. A aplicação foi construída utilizando a API LeJOS.

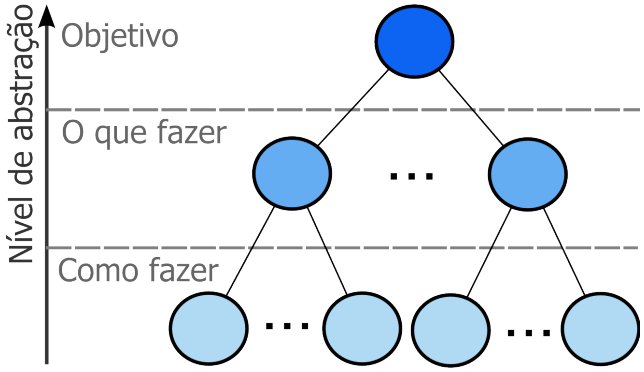


Figura 4.3: Ilustração da estruturação em árvore dos requisitos

4.2 Atividade "Identificar declarações do domínio"

Essa é a principal atividade do processo pois determina os elementos fundamentais da linguagem, sua abrangência, flexibilidade, entre outras características. Nela, o engenheiro de domínio extrai e classifica declarações do domínio a partir dos requisitos da aplicação de entrada. A atividade é dividida em duas subatividades que são descritas abaixo.

4.2.1 Subatividade "Extrair declarações do domínio a partir dos requisitos"

Constitui-se da estruturação dos requisitos da aplicação em formato de árvore, como pode ser visto na Figura 4.3. O primeiro nó representa o principal objetivo da aplicação. O segundo nível é chamado de "O que fazer" e decompõe o primeiro nó em outros nós que descrevem as funções que o robô deve executar para atingir o objetivo. Ainda, esses nós podem ter nós filhos que sejam mais descritivos e que continuam pertencendo ao mesmo nível. Pode-se pensar em nós filhos como descrições com menores níveis de abstração que seus nós pais. Esses nós devem descrever funções que não possuam referências a detalhes de baixo nível.

O terceiro nível é chamado de "Como fazer" e contém nós que descrevem como executar as funções descritas pelos nós do nível superior. A descrição dessas funções devem estar relacionadas à forma como os componentes do robô (por exemplo, sensores, atuadores, dispositivos, etc.) são utilizados para executá-las. Deve-se tomar cuidado para que os nós do terceiro nível não contenham detalhes específicos de programação, mas apenas declarações que relacionem os componentes. Por exemplo, considere que um nó do segundo nível tenha a descrição "andar seguindo paredes"; ligado a esse pode-se ter três nós no terceiro nível cujas descrições sejam "acionar motores de locomoção para frente", "usar sensor ultrassônico para medir distância à parede" e "alinhar com o algoritmo PID". O engenheiro de domínio pode convenientemente

omitir dos nós detalhes considerados óbvios; por exemplo, na descrição do primeiro nó do terceiro nível é possível omitir os motores, deixando apenas "andar para frente" (pois pressupõe o uso de motores de locomoção).

Com a árvore criada, o engenheiro de domínio deve escolher dentre os nós do segundo e terceiro níveis quais declarações serão consideradas para construir a linguagem desejada. Se os nós folhas forem escolhidos, a linguagem resultante será mais flexível porém mais verbosa e menos relacionada com o domínio. Porém, se apenas os nós do segundo nível forem selecionados, a linguagem resultante será mais relacionada com o domínio e será mais expressiva, no entanto não terá muita flexibilidade e será mais limitada. Também é possível selecionar nós de ambos os níveis. De modo geral, quanto mais abstrato for o nó, menos flexível serão os elementos da linguagem relacionados a ele.

4.2.2 Subatividade "Classificar as declarações do domínio em interesses"

As declarações do domínio selecionadas na subatividade anterior devem ser classificadas em *interesses*. Esses interesses são característicos do domínio da robótica móvel e elicitar os mais adequados pode depender da experiência com desenvolvimento de robôs móveis. Alguns interesses são elementares, como movimentação, detecção, comunicação, e assim por diante; porém, aplicações maiores e complexas podem resultar em interesses mais específicos de domínio.

A intenção dessa classificação é que cada interesse agrupe declarações que são fortemente relacionadas. Isso permite que os elementos da linguagem resultante sejam coesos e bem definidos. O resultado dessa atividade é o artefato Tabela de Declarações, a qual contém as declarações do domínio devidamente agrupadas por interesses.

Para o desenvolvimento da DSL 1, durante essa atividade foram estruturados os requisitos da aplicação feita na atividade anterior como ilustrado na Figura 4.4. Dividiu-se o objetivo principal "Monitorar portas de um corredor" em três nós: "Andar pelo corredor", "Detectar portas abertas", e "Emitir som caso a porta esteja aberta". (Nota: poderia haver uma aplicação em que seus nós fossem bem diferentes desses mesmo que o objetivo principal fosse o mesmo.)

Como definido no processo, o nível "Como fazer" tem de estar relacionado aos sensores, atuadores e outros possíveis dispositivos da plataforma robótica alvo. Portanto, foi definido que, para o robô andar pelo corredor, deve acionar seus motores de locomoção para frente, usar o sensor de toque para verificar o fim do corredor e dar meia-volta. Para detectar as portas abertas, ele deve usar um sensor ultrassônico para medir a distância entre o robô e as paredes e portas.

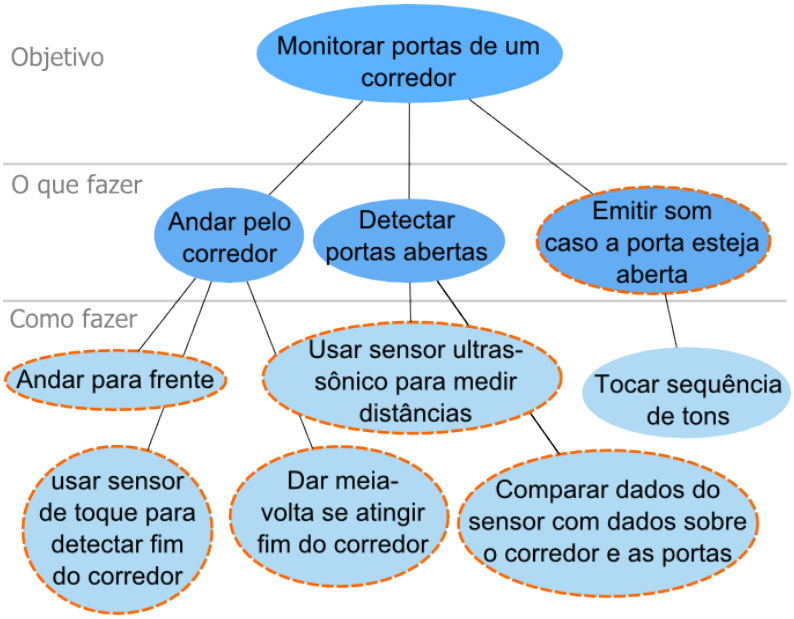


Figura 4.4: Requisitos da aplicação estruturados como árvore

Interesse	Declarações
Movimentação	Andar para frente
	Dar meia-volta se atingir fim do corredor
Detecção	Usar sensor de toque para detectar fim do corredor
	Usar sensor ultrassônico para medir distâncias
	Comparar dados do sensor com dados sobre o corredor e as portas
Comunicação	Emitir som caso a porta esteja aberta

Tabela 4.1: Relação dos interesses da DSL 1 e suas declarações

O robô deve possuir informações sobre o corredor e suas portas para poder confrontá-las com os dados obtidos pelo sensor. E, por fim, um alerta sonoro é emitido caso uma porta aberta seja identificada, e esse alerta é apenas uma sequência de tons.

Depois da extração das declarações, é necessário escolher quais delas farão parte da construção da DSL. É possível, como dito anteriormente, selecionar declarações de diferentes níveis. As declarações selecionadas são as que possuem contornos tracejados na Figura 4.4. Como o objetivo era construir uma DSL flexível, todos os nós folhas foram escolhidos, exceto o nó "Tocar sequência de tons"–ao invés, foi selecionado seu nó superior, para exemplificar o resultado da escolha desse tipo de nó.

Para finalizar essa atividade, é preciso categorizar as declarações escolhidas em interesses. Na Tabela 4.1 estão relacionados os interesses selecionados e suas respectivas declarações.

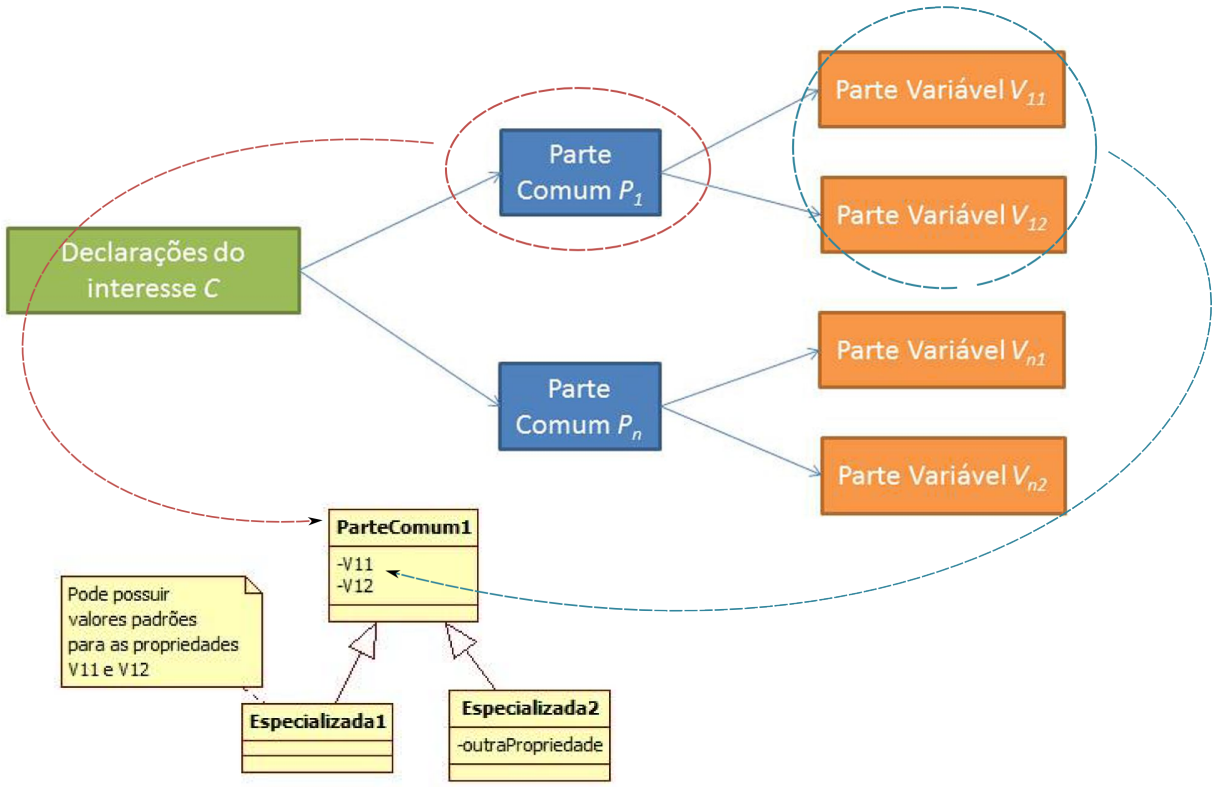


Figura 4.5: Decomposição das declarações em partes comuns e variáveis

4.3 Atividade "Criar modelo da linguagem"

Com a Tabela de Declarações construída na atividade anterior, o engenheiro de domínio deve, para cada interesse, extrair elementos que irão compor a linguagem, ou seja, entidades, relacionamentos e propriedades. Esses elementos irão compor o *modelo da linguagem* ou *metamodelo*. Para extrair esses elementos, é preciso identificar as partes comuns e variáveis das declarações que estão no interesse em questão.

Como exemplo de identificação das partes comuns e variáveis, considere declarações como "ir para frente", "ir para trás", "rotacionar" e "fazer curvas". Uma parte comum é que todas acionam os motores de locomoção. Dentre as partes variáveis estão direção, velocidade, distância e ângulo. Na Figura 4.5 há uma ilustração das partes comuns e variáveis de um determinado interesse C. Nela, essas declarações possuem as partes comuns $P_1..P_n$, as quais possuem as partes variáveis $V_{p1}..V_{pn}$ para cada parte comum p. Normalmente, as partes comuns são transformadas em entidades da DSL, enquanto que as partes variáveis, em propriedades dessas entidades. Pode-se estender essas entidades criando especializações com propriedades adicionais ou que tenham algumas propriedades com valores já definidos.

Com base na experiência obtida com a realização deste projeto de mestrado, foi criado um

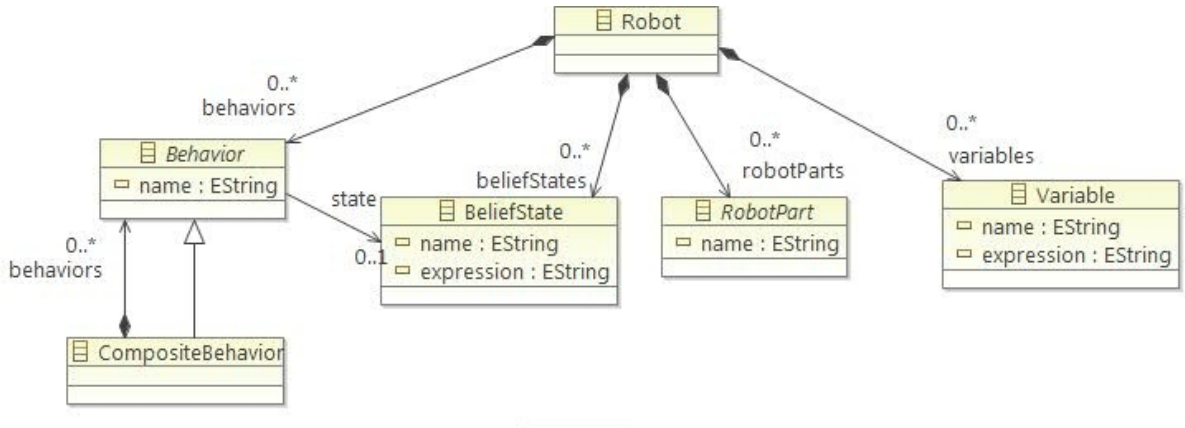


Figura 4.6: Modelo de linguagem genérico que serve de base para novas DSLs

modelo de linguagem genérico que pode ser usado como base para novas DSLs para robôs móveis. O engenheiro de domínio pode simplesmente estender esse modelo de linguagem para atender suas necessidades. Esse modelo é mostrado na Figura 4.6.

A entidade principal é a *Robot*, a qual representa a aplicação robótica. Ela é composta de zero ou mais instâncias das entidades *Behavior*, *BeliefState*, *Variable* e *RobotPart*. Essa configuração tem como base a arquitetura de subsunção, introduzida por Brooks (1986). Nela, o funcionamento do robô é dividido em vários comportamentos elementares ou simples (*behaviors*) cujas execuções são mutuamente exclusivas. Cada um desses comportamentos possui uma condição que precisa ser satisfeita para que ele seja executado.

Essa condição é representada pela propriedade *expression* da entidade *BeliefState.* Essa contém comparações entre dados provenientes de dispositivos do robô (entidade *RobotPart*), como sensores, e valores de variáveis (instâncias da entidade *Variable*). A expressão também pode utilizar funções dos dispositivos e também das variáveis. Por exemplo, um sensor ultrassônico pode fornecer a função "distance" que retorna um valor *float*. Inclusive, o engenheiro de domínio pode criar funções especializadas que utilizam algoritmos complexos comumente utilizados na robótica, como o filtro gaussiano, o algoritmo Proporcional-Integral-Derivativa (PID), entre outros. A entidade *CompositeBehavior* permite agrupar comportamentos para que eles sejam executados sequencialmente ao invés de serem regidos pelas suas condições.

Usando esse modelo genérico, o engenheiro de domínio pode desenvolver sua própria DSL criando novas entidades que especializam as já existentes. A alternativa mais simples é criar comportamentos especializados, ou seja, entidades que estendam *Behavior*, e criar entidades que representem os sensores que estão disponíveis na plataforma alvo, estendendo a entidade *RobotPart*. O engenheiro também é livre para criar novas propriedades dentro das entidades existentes ou mesmo alterar as que estão presentes.

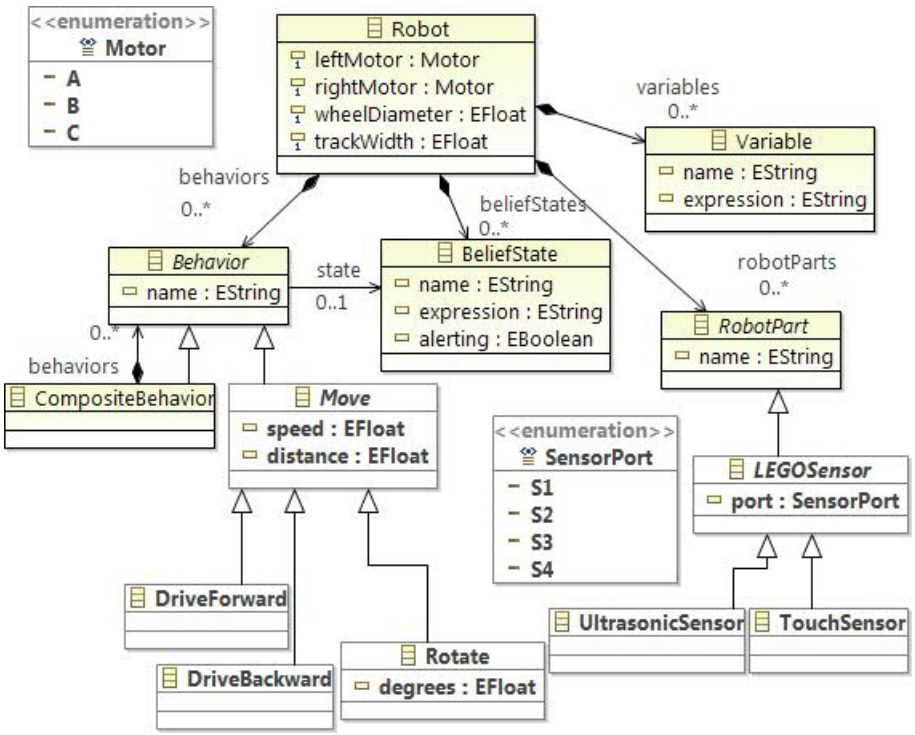


Figura 4.7: O modelo de linguagem da DSL 1

Uma DSL também pode ter diferentes tipos de diagrama, assim como a UML possui diagramas de classe, de sequência, de atividades, etc. O engenheiro de domínio pode analisar o modelo da linguagem para verificar se todas as entidades podem ser representadas em um único diagrama. Dois casos em que duas entidades podem ser separadas em diagramas diferentes são quando elas são mutuamente independentes ou quando juntas podem resultar em modelos poluídos, de difícil leitura. Deve haver um balanço entre entidades correlatas e poluição dos modelos, ou seja, mesmo que todas as entidades de um diagrama sejam bem relacionadas, isso pode resultar em modelos poluídos.

Nessa atividade, o modelo de linguagem genérico supracitado foi utilizado para desenvolver a DSL 1 e o resultado pode ser visto na Figura 4.7. As novas entidades criadas estão em negrito. Também foram criadas quatro novas propriedades na entidade *Robot* e uma na entidade *BeliefState*.

O primeiro interesse a ser analisado foi o Movimentação. A declaração "Andar para frente" foi transformada na entidade *DriveForward*, e a "Dar meia-volta se atingir o fim do corredor" foi transformada em duas: *DriveBackward* e *Rotate*. Todas essas entidades precisavam de dois motores para realizar suas funções, e esses motores poderiam ser conectados a qualquer das três portas A, B e C do microcontrolador. Os motores são especificados nas propriedades *leftMotor* e *rightMotor* e as portas são representadas pela enumeração *Motor*. Após refatorações, as entidades *DriveForward*, *DriveBackward* e *Rotate* foram generalizadas na entidade abstrata *Move*,

e a especificação dos motores foi deslocada para a entidade *Robot*.

Na análise do interesse Detecção, suas declarações se resumiam a combinações de dados dos sensores com dados de variáveis internas e, por isso, foi definido que as entidades existentes *BeliefState* e *Variable* são suficientes. Foi preciso somente estender a entidade *RobotPart* com representações dos sensores utilizados pela aplicação, resultando nas entidades *UltrasonicSensor* e *TouchSensor*. A entidade abstrata *LEGOSensor* foi resultado de uma refatoração–todos os sensores podem ser conectados às portas S1, S2, S3 e S4 do microcontrolador. Por isso também foi criada a enumeração *SensorPort*.

O interesse Comunicação possui a declaração de emissão de som. Essa representa uma ação diferente das demais porque ela é executada concorrentemente com os demais comportamentos de movimentação. Então foi decidido criar uma nova propriedade booleana chamada *alerting* na entidade *BeliefState*. Dessa forma, é possível dizer à aplicação que soe um alerta quando o robô estiver num determinado estado (por exemplo, quando encontrou uma porta aberta). Note que essa representação é limitada e menos flexível – não é possível especificar diferentes sequências de tons e durações. Isso aconteceu porque o nó que representa essa abstração não é um nó folha mas faz parte do nível "O que fazer".

Terminado o modelo, chegou-se à conclusão de que se todas as entidades fossem representadas em um único diagrama, poderiam haver modelos com grandes quantidades de elementos, o que limitaria a legibilidade e compreensão. Por isso, analisou-se quais entidades poderiam ser separadas para formar um outro tipo de diagrama. Como resultado, foram obtidos dois tipos de diagrama: o modelo de comportamentos (*Behavior Model*) e o modelo de estados de crença (*BeliefState Model*). No primeiro são especificados os comportamentos e, no segundo, os estados de crença junto com os sensores e as variáveis.

4.4 Atividade "Criar templates de geração de código"

Um template de código é um artefato que possui basicamente porções de código fixo e pontos de variação em que diferentes linhas de código podem ser inseridas. Essas peças são definidas pelo modelo especificado por uma DSL. O gerador de código interpreta o modelo e aloca as peças corretas nos pontos de variação. É similar a páginas PHP, que possuem código HTML fixo e comandos que geram conteúdo dinâmico.

Essa atividade está presente na maioria (se não todos) dos processos de construção de DSLs que pressupõem geração de código. A execução da mesma é bem parecida nos processos: deve-se ter um modelo construído com a DSL e seu equivalente em código; a partir disso, deve-se

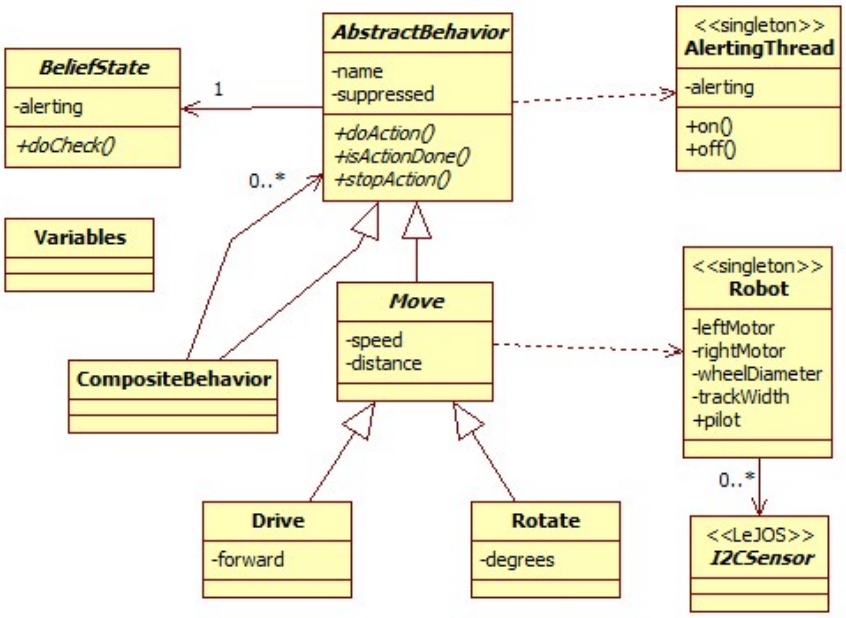


Figura 4.8: A arquitetura base das aplicações geradas pela DSL

identificar quais partes sempre estarão presentes nas aplicações geradas e quais são resultado do que está no modelo.

Não há necessidade do gerador produzir todo o código da aplicação. Códigos que se repetem e também estruturas do domínio podem constituir um framework de domínio, servindo de base para o código gerado e facilitando a tarefa do gerador.

Para criar os templates da DSL 1 foi utilizado o framework JET (*Java Emitter Templates*). Uma parte significativa do código da aplicação de entrada foi reutilizado e generalizado para atender às necessidades da DSL. Na Figura 4.8 é apresentada a arquitetura base de todas as aplicações que podem ser geradas pela DSL. Os templates criados geram essa arquitetura com as devidas modificações requeridas pela aplicação modelada, além de criar novas classes e objetos.

A classe *AbstractBehavior* é a classe pai de todos os comportamentos e possui o código necessário para integrá-los à aplicação. Cada comportamento especializa essa classe com a funcionalidade desejada. Instâncias de *DriveFoward* e *DriveBackward* na modelagem resultam em objetos da classe *Drive*. Cada instância de *BeliefState* na modelagem é transformada em uma classe que estende a classe *BeliefState*. Instâncias de *Variable* são transformadas em atributos da classe *Variables*.

Como um exemplo de como é basicamente o funcionamento dos templates, seja o fato de que, durante a modelagem, o desenvolvedor pode especificar as propriedades do robô, ou seja,

Listagem 4.1: Uma parte do template Robot.jet que gera a classe Robot

```
1  public class Robot {
2
3    private NXTRegulatedMotor leftMotor =
4      Motor.<c:get select="/contents/@leftMotor"/>;
5    private NXTRegulatedMotor rightMotor =
6      Motor.<c:get select="/contents/@rightMotor"/>;
7    private float wheelDiameter =
8      <c:get select="/contents/@wheelDiameter"/>f;
9    private float trackWidth =
10     <c:get select="/contents/@trackWidth"/>f;
11
12   // demais códigos omitidos
13 }
```

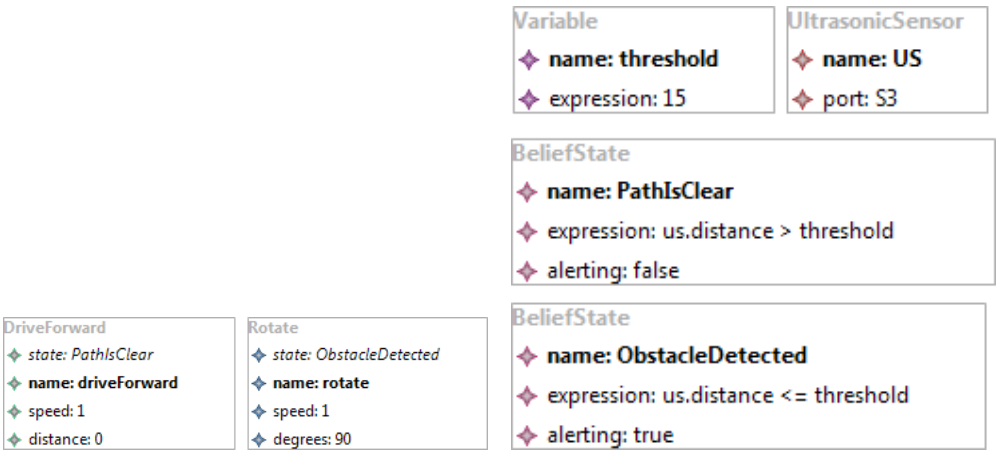
Listagem 4.2: Exemplo de geração da classe Robot

```
1  public class Robot {
2
3    private NXTRegulatedMotor leftMotor = Motor.A;
4    private NXTRegulatedMotor rightMotor = Motor.B;
5    private float whellDiameter = 2.2f;
6    private float trackWidth = 5.6f;
7
8    // demais códigos omitidos
9  }
```

as portas de seus motores de locomoção (*leftMotor* e *rightMotor*), o diâmetro de suas rodas (*wheelDiameter*) e a distância entre as mesmas (*trackWidth*). Essas e outras informações são coletadas pelo template *Robot.jet* que gera o arquivo *Robot.java*. Na Listagem 4.1 é apresentada uma parte do template *Robot.jet*. Nele há código fixo em Java e comandos específicos do JET. Esses comandos retornam os valores informados na modelagem. Supondo que o desenvolvedor informou que as portas dos motores de locomoção são A e B (esquerdo e direito), o diâmetro das rodas é de 2,2 cm e o comprimento do eixo é de 5,6 cm, a classe Robot a ser gerada será a exemplificada na Listagem 4.2.

4.5 Atividade "Criar a notação da linguagem"

A notação da linguagem é a sua sintaxe concreta, ou seja, como as entidades, propriedades e relacionamentos são representados visualmente pela DSL. A sintaxe concreta é formada por: um conjunto de símbolos gráficos (elementos 1D, 2D, 3D, textos, etc.) chamado de vocabulário



(a) Modelo de comportamentos (b) Modelo de estados de crença

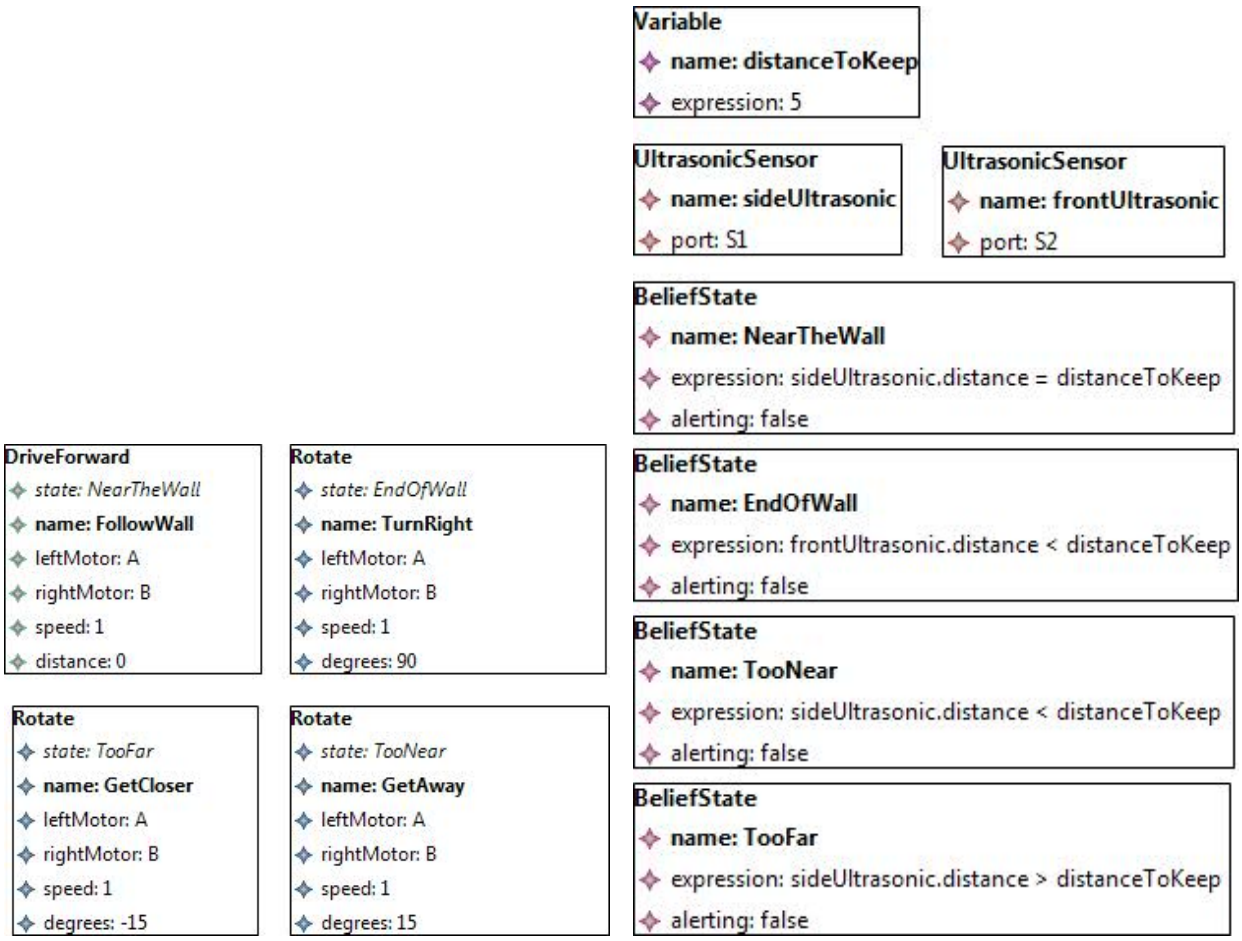
Figura 4.9: Modelos da aplicação de desvio de obstáculos

visual e um conjunto de regras de composição chamado de gramática visual.

Moody (2009) define um conjunto de princípios para projetar notações visuais que são cognitivamente efetivas. Eles podem ser utilizados não só para criar notações mas também para avaliar, comparar e evoluir notações já existentes. Dentre os princípios está o de Claridade Semiótica constatando que deve haver uma correspondência um para um entre os elementos da linguagem e os símbolos gráficos, sob pena de ocorrer redundância de símbolos, sobrecarga, excesso ou mesmo déficit. Outro princípio particularmente importante para se aplicar a DSLs de sistemas embarcados é o de Gerenciamento da Complexidade. Isso é a habilidade de representar informações sem sobrecarregar a capacidade humana de compreendê-las. Esse princípio prevê a inclusão de mecanismos explícitos para lidar com complexidade. Tais mecanismos devem permitir a modularização e estruturação hierárquica das notações. Segundo Moody (2002) e Nordbotten e Crosby (apud MOODY, 2009) tanto a modularização quando a estruturação hierárquica podem melhorar a compreensão dos diagramas por parte dos usuários finais.

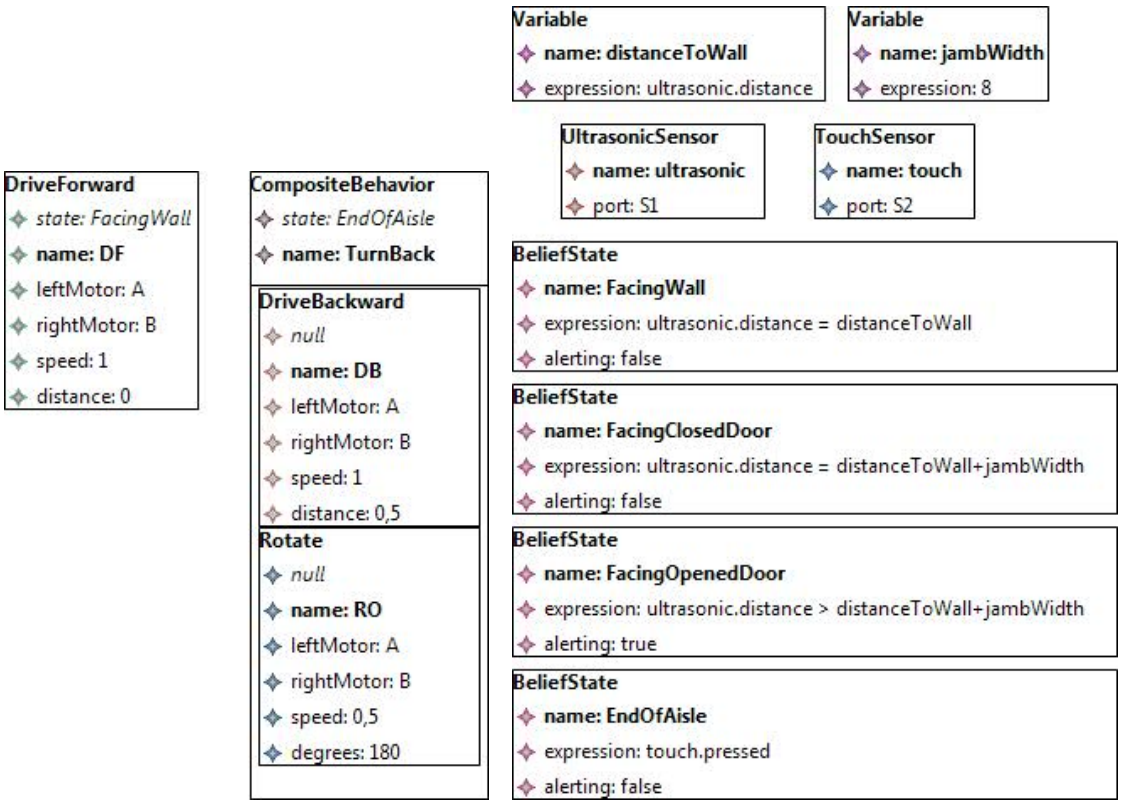
Nesta atividade, foi criada a representação gráfica dos elementos da DSL 1. Considerou-se que escolher a notação mais adequada para essa DSL estaria fora do escopo da prova de conceito e, por isso, foi decidido utilizar retângulos para representar todas as entidades diferenciando-as entre si com uma etiqueta em negrito contendo seu tipo por escrito. As propriedades das entidades seguem o padrão *<nome> : <valor>* e aparecem listadas abaixo dessa etiqueta.

Embora o processo tenha como entrada uma única aplicação quando muitos processos de engenharia de domínio baseiam-se em três ou mais, aplicações diferentes da de entrada podem ser construídas com a DSL resultante. Nesta seção são apresentadas algumas aplicações que podem ser feitas com a DSL resultante.



(a) Modelo de comportamentos (b) Modelo de estados de crença

Figura 4.10: Modelos da aplicação do seguidor de paredes



(a) Modelo de comportamentos (b) Modelo de estados de crença

Figura 4.11: Modelos da aplicação de monitoramento de portas

Na Figura 4.9 são mostrados os modelos de uma aplicação robótica para desvio de obstáculos. O robô vagueia e se rotaciona 90º quando encontra um obstáculo a menos de 16 centímetros de sua frente. O comportamento *driveForward* é acionado quando o estado de crença do robô for *PathIsClear*, e *rotate* é acionado quando o estado é *ObstacleDetected*. Na expressão do estado *PathIsClear* há a comparação entre a distância medida pelo sensor ultrassônico *us*, obtida pela função *distance*, com o valor da variável *threshold*, que determina a distância máxima que o robô deve interpretar como obstáculo. No outro estado há situação semelhante, exceto pelo valor da propriedade *alerting*, indicando que o robô emitirá um aviso sonoro toda vez que entrar no estado *ObstacleDetected*.

Na Figura 4.10 são mostrados os modelos da aplicação do robô seguidor de paredes. O objetivo desse robô é andar perto da parede de uma sala contornando-a. Como o chão possui pequenas falhas e os motores de locomoção não são precisos, ocorre eventualmente um desalinhamento que aproxima ou afasta o robô da parede e, portanto, é necessário corrigi-lo. Por isso, há os comportamentos *GetCloser* e *GetAway*, que ajustam em 15 graus para a direita ou para a esquerda quando o estado de crença é *TooFar* ou *TooNear*, respectivamente. Nessa aplicação, o robô usa dois sensores ultrassônicos: um na parte frontal para detectar o fim da parede e um na

lateral para manter-se alinhado à ela, e são representados respectivamente por *frontUltrasonic* e *sideUltrasonic*. A varíavel *distanceToKeep* é utilizada para estabelecer a distância ideal que o robô deve manter da parede.

Finalmente, na Figura 4.11 pode-se ver os modelos da mesma aplicação de monitoramento de portas utilizada como entrada para criar essa DSL. O que há de novo nesses modelos é a utilização do sensor de toque, nomeado *touch*, e da sua função *pressed* na expressão do estado de crença *EndOfAisle*. Esse é o estado do robô quando o sensor de toque é pressionado, culminando no acionamento do comportamento composto *TurnBack*. Esse possui dois comportamentos que são executados em série: *DriveBackward*, que afasta o robô da parede em frente, e *Rotate*, que o rotaciona em 180º para continuar a varredura.

Capítulo 5

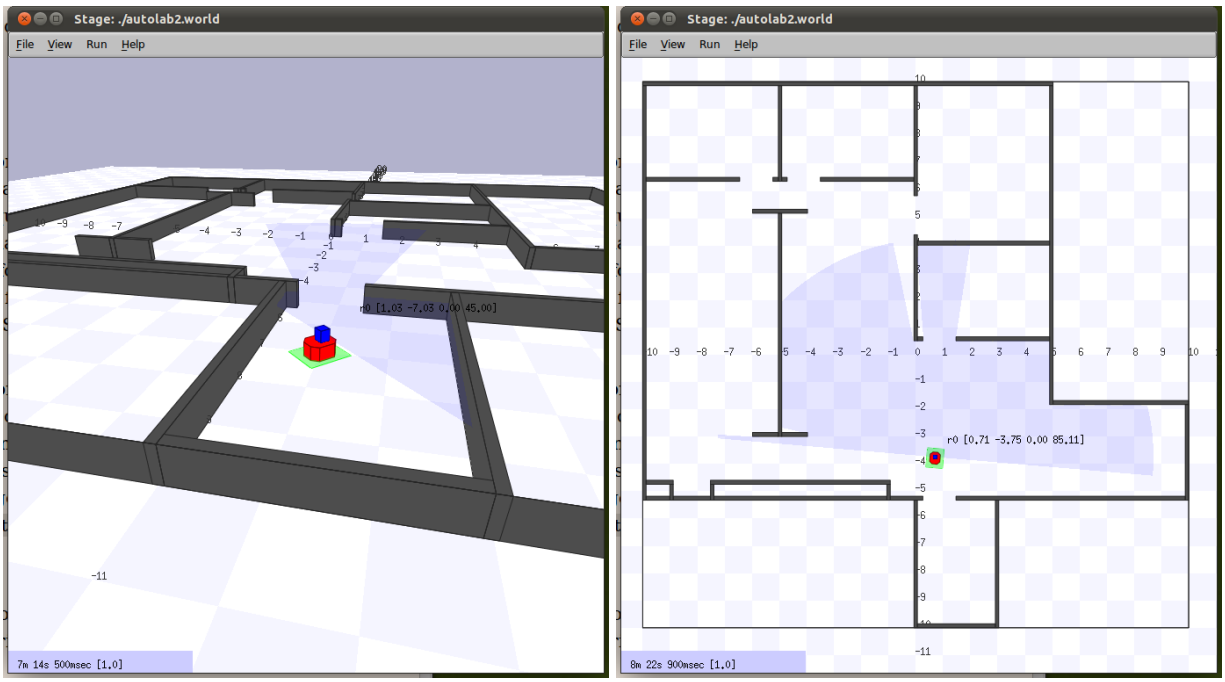
PROVA DE CONCEITO: ROBÔ ENTREGADOR DE MATERIAIS

Neste capítulo é apresentada uma prova de conceito em que envolve a criação de uma DSL a partir de um robô entregador autônomo de materiais, e que foi nomeada de DSL 2.

5.1 "Selecionar ou construir uma aplicação"

A prova de conceito foi feita utilizando como aplicação base um robô entregador de materiais, cujo código-fonte pode ser visto no Apêndice B. A função desse robô era a de percorrer um determinado ambiente com o objetivo de entregar de uma sala a outra materiais diversos, como remédios, comida, água, entre outros. O robô guardava em memória um mapa do ambiente para que pudesse conhecer as salas e traçar rotas. Além disso, qualquer obstáculo imprevisto deveria ser desviado. Em cada sala, havia um computador capaz de enviar ao robô comandos para que ele se movesse até alguma outra sala. Caso o comando enviado fosse a ida a uma sala desconhecida, o robô "dizia" usando um sintetizador de voz que desconhece tal sala. Essa aplicação foi simulada com o software Stage, em que o robô utilizado era do modelo Pioneer 2-DX, equipado com o sensor laser Sick LMS-200. Na Figura 5.1 há duas capturas de tela contendo a simulação dessa aplicação. Na parte (a) é possível visualizar em perspectiva uma simulação do robô Pioneer equipado com o sensor Sick. Na parte (b) é possível ver o mapa completo do ambiente e o alcance do feixe do sensor.

O software do robô é composto de três artefatos: um arquivo .world, que descreve o ambiente virtual, o robô e seus sensores; um arquivo .cfg, que configura os drivers que serão utilizados para controlar os dispositivos do robô; e um arquivo .cc, um código em C++ contendo a lógica da aplicação. Alguns dos drivers fornecidos pelo Player são considerados abstratos; ao



(a) Simulação do robô entregador em perspectiva (b) Simulação do robô entregador–vista aérea

Figura 5.1: Simulação do robô entregador visto de dois ângulos diferentes

invés de hardware, eles usam outros drivers para receber informação e enviar comandos. Representam algoritmos de localização, de planejamento, etc. que podem ser facilmente reutilizados. Nessa aplicação, por exemplo, são usados como drivers o algoritmo de localização adaptativa de Monte Carlo (FOX, 2001), o algoritmo de navegação Vector Field Histogram ou VFH (ULRICH; BORENSTEIN, 1998) e o algoritmo de planejamento de rotas Wavefront (LATOMBE, 1991).

5.1.1 “Identificar as abstrações do domínio”

Na Figura 5.2 são apresentados os requisitos da aplicação devidamente estruturados em árvore. Como o objetivo é entregar materiais quaisquer entre diferentes salas, foram criadas no nível “O que fazer” quatro declarações que refletem o que é necessário, em termos de funcionalidades do robô, para atingir esse objetivo. Para que ele entregue corretamente, é necessário um conhecimento prévio do ambiente, quais são as salas, onde elas ficam, e onde o robô se encontra no mapa em determinados momentos. Somente depois de ter todas essas informações é que o robô poderá locomover-se até a sala de destino. Os nós do nível “Como fazer” são declarações que envolvem funcionalidades do software Player e dispositivos robóticos. Todas essas foram extraídas dos três artefatos que compõem a aplicação.

Apenas os nós folhas foram selecionados para comporem a DSL pois o objetivo era maximizar a sua flexibilidade. A segunda etapa dessa atividade é a classificação desses nós em

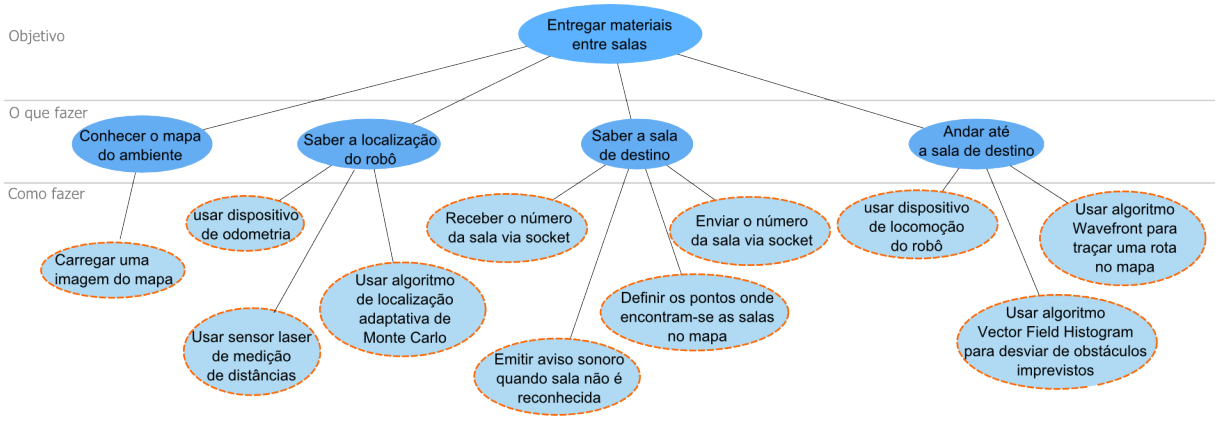


Figura 5.2: Estruturação em árvore dos requisitos da aplicação entregadora de materiais

interesses. Foram identificados os interesses *Localização, Controle* e *Comunicação*. A relação de suas declarações pode ser vista na Tabela 5.1.

5.1.2 "Criar o modelo da linguagem"

Nesta atividade, os interesses são analisados, coletam-se partes comuns e variáveis e constrói-se o modelo da linguagem ou metamodelo, que pode ser visto na Figura 5.3. Foi utilizado como base o modelo genérico descrito na Seção 4.3.As novas entidades criadas na atividade estão destacadas com fundo branco e fonte em negrito. As entidades e relacionamentos que estão apenas em negrito são modificações das entidades e relacionamentos originais do modelo genérico.

A criação do modelo começou pela análise do interesse *Localização*. Analisando as declarações à luz do código-fonte, percebeu-se que a maioria delas tem em comum a referência a drivers e dispositivos do robô. No Player, esses dispositivos podem ser controlados por meio de interfaces. Uma interface, no Player, especifica como uma determinada classe de dispositivos podem ser controlados. A interface *position2d*, por exemplo, é destinada a robôs móveis terrestres e permite-os receber comandos para se moverem ou reportar o seu estado (por exemplo, velocidade e posição). O que faz um dispositivo suportar uma interface é um *driver*[1].

A declaração "Carregar uma imagem do mapa", por exemplo, faz alusão ao driver *mapfile*, que fornece a interface *map*. Já a declaração "Usar dispositivo de locomoção do robô" deveria referenciar o driver do robô Pioneer, *p2os*, mas como a aplicação é simulada, ela referencia *stage*, que é o driver simulador. Chegou-se então à definição de uma entidade abstrata *Driver* (parte comum) e de uma entidade concreta para cada driver referenciado, com suas próprias propriedades (partes variáveis): *Mapfile*, *MonteCarlo, StageRobot* e *StageSimulation*. *Driver* estende de *RobotPart* porque agrega tanto a interface quanto a implementação de um dispositivo

[1]Fonte: http://playerstage.sourceforge.net/doc/Player-3.0.2/player/

Interesse	Declaração
Localização	Carregar uma imagem do mapa
	Usar algoritmo de localização adaptativa de Monte Carlo
	Usar dispositivo de odometria
	Usar sensor laser para medir distâncias
	Usar mapa
	Definir os pontos onde encontram-se as salas no mapa
Controle	Usar algoritmo wavefront para traçar uma rota no mapa
	Usar dispositivo de locomoção do robô
	usar algoritmo Vector Field Histogram para desviar de obstáculos
Comunicação	Emitir aviso sonoro para sala não reconhecida
	Receber o número de salas via socket
	Enviar o número da sala via socket

Tabela 5.1: Relação dos interesses da DSL 2 e suas abstrações

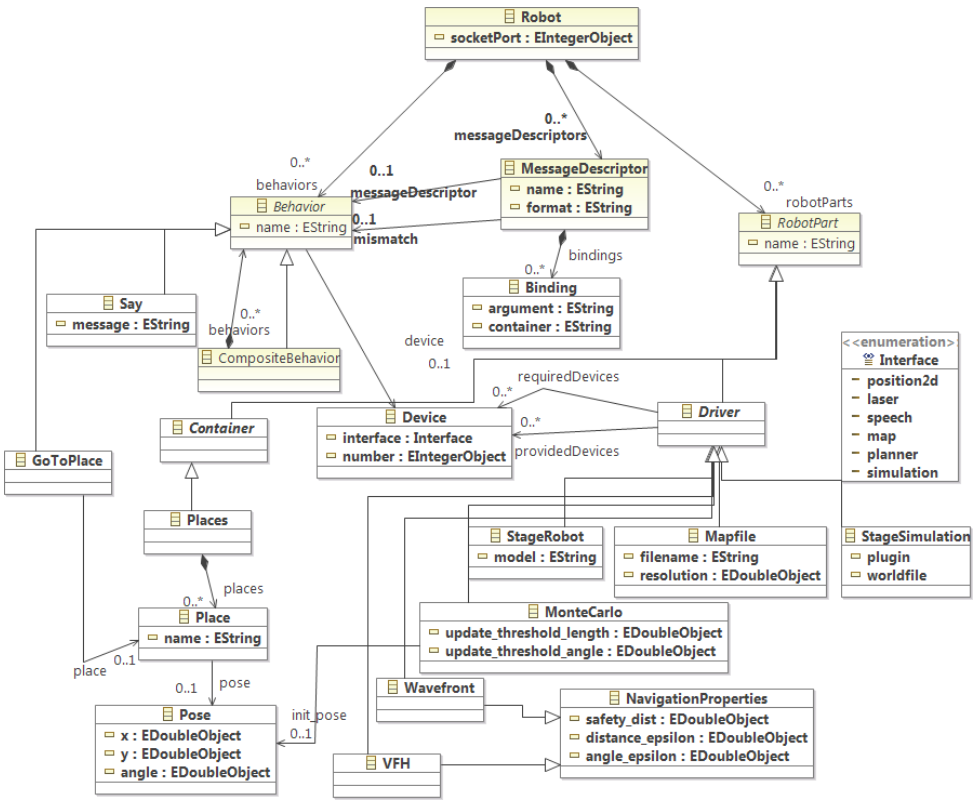


Figura 5.3: Robô entregador de materiais–modelo da linguagem

(ou uma peça) do robô. A entidade *Device* foi criada para representar dispositivos. Um *Device* é definido por uma interface e um número identificador, que o diferencia de outros dispositivos com mesma interface. As interfaces são um conjunto finito e, portanto, são melhor representadas por uma enumeração (entidade Interface). Um driver pode fornecer (ou implementar) mais de um dispositivo, assim como pode requerer informações de outros dispositivos para que funcione corretamente. A partir disso, criou-se dois relacionamentos um para muitos entre *Driver* e *Device*, a saber, *providedDevices* (os dispositivos que o *Driver* implementa) e *requiredDevices* (os dispositivos que o *Driver* necessita). A única declaração que não é relacionada a driver envolve a definição de pontos fixos no mapa representando lugares que devem ser conhecidos. Por isso, foram criadas as entidades *Places*, *Place* e *Pose*. *Places* é um contêiner de objetos *Place,* os quais são como lugares nomeados no mapa (por exemplo, a sala de remédios, sala de cirurgia, etc.). A entidade *Pose* representa um ponto no mapa com as coordenadas *x* e *y*, além do ângulo para a parte frontal do robô.

O próximo e último interesse analisado fora *Comunicação*. O que há de comum entre as suas declarações é o fato de serem funções (ou comportamentos) que o robô pode executar, ou seja, emitir um som ou realizar determinada tarefa ao receber alguma informação via socket. Dentro do código-fonte, esses comportamentos são chamadas a métodos de objetos que representam os dispositivos do robô. Por isso, criou-se então um relacionamento entre *Behavior* e *Device*. Também foram criados entidades para representarem esses comportamentos, as quais são *Say* e *GoToPlace,* ambas estendendo *Behavior*. Os parâmetros relevantes e passíveis de variação que eram passados a esses métodos foram traduzidos em propriedades dessas entidades.

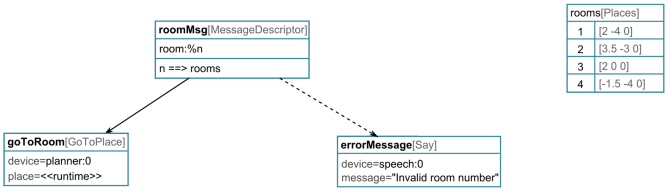
O comportamento do robô é alterado quando este recebe alguma informação via socket. Nesse caso, a entidade *BeliefState* seria a responsável por receber e decodificar essa informação, pois é ela quem controla os comportamentos. Uma das variabilidades está em justamente receber diferentes tipos de informação, e não somente as coordenadas da sala de destino. Por isso, os *beliefStates* podem atuar como descritores de mensagens, ou seja, fazerem a ligação entre um determinado tipo de mensagem e os comportamentos que devem processá-lo. Dessa forma, porém, a arquitetura de subsunção dá lugar a uma abordagem orientada a eventos (ou mensagens) e a entidade *BeliefState* muda de nome para *MessageDescriptor*. A propriedade *expression* passa a ser *format* que, nos moldes da função *printf()* da linguagem C, formata a mensagem proveniente do socket por meio de argumentos. Por exemplo, se para informar a sala de destino do robô as mensagens recebidas seguem o padrão `sala:<numero>`, a propriedade *format* pode ter o valor `sala:%n`, em que *n* é um argumento que será captado pelo descritor de mensagens e passado ao comportamento ligado a ele. Como recurso adicional, pode-se realizar um mapeamento entre o argumento e os objetos de um contêiner. Por exemplo, o argumento *n*

pode ser utilizado para realizar uma busca em um contêiner *Places* e enviar ao comportamento um objeto *Place* ao invés de um simples número. Para isso, foi criada a entidade *Binding*, que mapeia um argumento da propriedade *format* a um contêiner criado pelo desenvolvedor. Caso a informação recebida não exista dentro do contêiner especificado, é acionado um comportamento alternativo, o qual é indicado pelo relacionamento *mismatch*. Por fim, foi criada a propriedade *socketPort* dentro da própria entidade *Robot* para que o desenvolvedor possa especificar a porta pela qual o socket da aplicação receberá as mensagens.

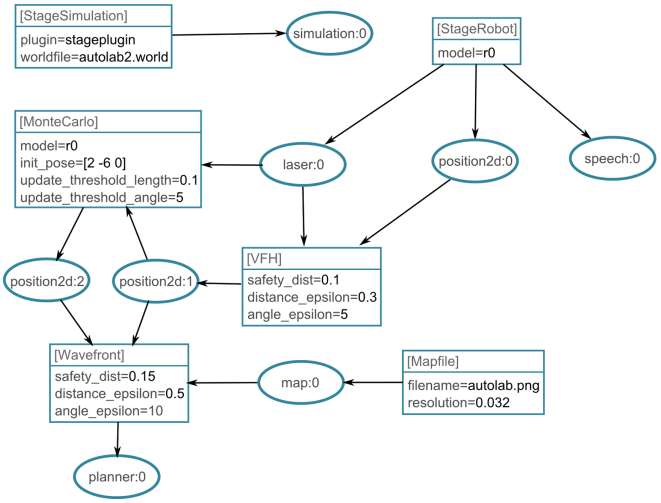
5.1.3 Aplicações modeladas com a DSL 2

Na Figura 5.4 é apresentada a mesma aplicação utilizada como entrada no processo modelada com a DSL resultante. A DSL possui dois tipos de diagramas: um para especificação dos comportamentos e outro para os dispositivos do robô. No diagrama de comportamentos (Figura 5.4 parte (a)), tem-se o descritor de mensagens *roomMsg* (*MessageDescriptor*) que interpreta as mensagens provenientes do socket, extrai delas o número da sala de destino, realiza uma busca no contêiner *rooms* (*Places*) e envia o resultado ao comportamento *goToRoom* (*GoToPlace*). Caso o número recebido não esteja dentro do contêiner, o comportameno *errorMessage* (*Say*) é acionado. Esse comportamento envia comandos ao dispositivo *speech:0*, que é um sintetizador de voz. Já o comportamento goToRoom usa o *planner:0*, um dispositivo virtual cujo driver implementa o algoritmo Wavefront.

O segundo tipo de diagrama é o de dispositivos do robô. Na Figura 5.4 parte (b) é apresentada a modelagem dos dispositivos do robô entregador. Os drivers estão representados em retângulos contendo suas propriedades e valores, e os dispositivos são representados por elipses contendo a interface implementada e o número identificador. Por exemplo, o dispositivo *speech:0*, utilizado pelo comportamento *errorMessage*, tem como driver o *StageRobot*, que representa o robô dentro do simulador. Esse mesmo driver oferece os dispositivos *laser:0*, representando o sensor Sick LMS200, e *position2d:0*, representando a parte de locomoção do robô. As setas que incidem sobre os dispositivos os ligam ao driver que os implementam, e as setas que incidem sobre os drivers os ligam aos dispositivos que esses necessitam. Por exemplo, o driver *VFH* utiliza dois dos dispositivos implementados por *StageRobot*, os quais são *laser:0* e *position2d:0*; e fornece o dispositivo *position2d:1*. Ou seja, os comandos enviados a esse dispositivo passarão primeiro para o driver *VFH*, que aplicará o seu algoritmo interno de desvios de obstáculos em cima das informações obtidas pelo *laser:0* e *position2d:0*, e enviará os comandos adequados ao *position2d:0*.

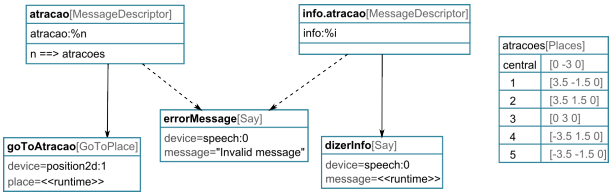


(a) Diagrama de comportamentos

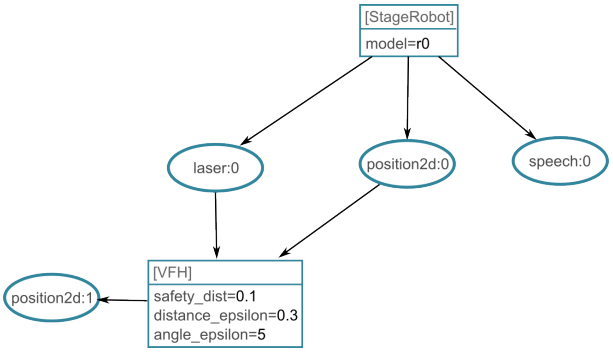


(b) Diagrama de dispositivos

Figura 5.4: Modelagem da aplicação do robô entregador de materiais



(a) Modelo de comportamentos



(b) Modelo de dispositivos

Figura 5.5: Modelagem da aplicação do robô guia de museu

Modelagem de uma aplicação para um robô guia de museu

Com essa mesma DSL foi modelada uma aplicação para um robô guia de museu, no intuito de mostrar a possibilidade de modelagem de aplicações diferentes da de entrada. O objetivo desse robô é percorrer as atrações de um museu que possui o formato de uma elipse. Um computador central é o responsável por enviar ao robô as atrações em que ele deve ir juntamente com informações sobre as mesmas. Na Figura 5.5, parte (a), é exibido o diagrama de comportamentos da aplicação. Há dois descritores de mensagens, *atracao* e *info.atracao*. O primeiro recebe do socket a atração na qual o robô deve ir. As coordenadas das atrações estão no contêiner *atracoes*. Já o segundo descritor recebe as informações da atração onde o robô se deslocou e aciona o comportamento *dizerInfo* passando a ele a informação recebida. Nesse descritor, não há mapeamento do argumento *i*, portanto, o próprio *i* é enviado ao comportamento. Ambos os descritores acionam o comportamento *errorMessage* caso a mensagem recebida pelo socket seja inválida.

Na parte (b) da Figura 5.5 é apresentado o diagrama de dispositivos. O dispositivo *position2d:1* é o acionado pelo comportamento *goToAtracao* (parte (a)). Como o driver desse dispositivo é o algoritmo *VFH*, o robô consegue chegar ao destino desviando dos possíveis obstáculos encontrados ao longo do caminho. E, da mesma forma que na aplicação anterior, o *VFH* envia comandos ao dispositivo *position2d:0*, que é o dispositivo de locomoção do robô simulado pelo Stage.

Capítulo 6

CONCLUSÃO

Este trabalho apresenta uma abordagem de desenvolvimento de linguagens de modelagem direcionadas ao domínio da robótica móvel. Aqui foram apresentadas duas DSLs como resultado da aplicação dessas especificações dentro do processo. A motivação para o desenvolvimento dessas especificações está no fato de que as aplicações robóticas são complexas, exigem um bom conhecimento sobre engenharia, física, mecânica, etc., e até o momento é considerado um domínio incomum de se encontrar na maioria das pesquisas em Engenharia de Software. Por causa disso, poucos são os trabalhos que apresentam diretrizes para desenvolver DSLs para esse domínio; na maioria dos processos impera o domínio de sistemas de informação e sua aplicação na robótica pode comprometer a qualidade não só da DSL mas também dos sistemas modelados com ela.

O caminho, apresentado neste trabalho para lidar com as questões do domínio da robótica, passa necessariamente pela consideração dos dispositivos de hardware e da forma como são empregados para atingir os objetivos da aplicação robótica. Após o levantamento e classificação de declarações sobre o domínio, aplica-se uma análise das comunalidades e das variabilidades; há ainda um mapeamento geral entre essas partes e os componentes de uma DSL, a saber, entidades, propriedades e relacionamentos. Destaca-se, ainda, a discussão sobre a criação de diferentes tipos de diagramas, cada um contendo conjuntos de componentes da linguagem que formam um tema (por exemplo, pode-se dizer que o tema do diagrama de classes da UML *são* classes), respeitando um equilíbrio entre entidades correlatas e poluição potencial dos modelos. Foi apresentado um modelo para estruturação de requisitos na forma de árvore, onde mostrou-se possível extrair elementos de uma DSL a partir de seus nós. Esse modelo distribui o grau de abstração por entre os níveis dos nós, o que permite selecionar a flexibilidade e expressividade da DSL. Para facilitar a construção de DSLs para a robótica, foi apresentado um metamodelo genérico com uma implementação da arquitetura de subsunção. O desenvolvedor pode criar

DSLs apenas estendendo as entidades desse modelo; ainda, o desenvolvedor também é livre para alterá-lo convenientemente, como aconteceu no desenvolvimento da DSL 2, em que algumas entidades tiveram seus nomes alterados e alguns relacionamentos refeitos para refletir melhor o domínio em questão.

A abordagem pode ser aplicada utilizando aplicações de diferentes complexidades, uma vez que o modelo de requisitos em árvore permite que nós relacionados estejam no mesmo nível; isso quer dizer que um determinado nó pode ser "quebrado" em nós menores que pertençam ao mesmo nível. Além disso, a classificação dos nós em interesses facilita a análise, pois ainda que uma determinada árvore possua uma grande quantidade de nós, eles são analisados em conjunto e não separadamente.

O desenvolvimento da DSL 1 foi utilizado para exemplificar a aplicação do processo de desenvolvimento de DSLs levando em consideração as especificações produzidas neste trabalho. Essa DSL é capaz de modelar aplicações para a plataforma LeJOS, um firmware para o Lego Mindstorms que permite o uso da linguagem Java. Já a DSL 2 é capaz de modelar aplicações que tenham como base o Player e o Stage, um software robótico cliente-servidor e um simulador 2D. Essas duas DSLs mostraram-se capazes de modelar aplicações diferentes das que foram utilizadas para o seu desenvolvimento. Isso permite afirmar que, com uma única aplicação base, é possível criar uma versão inicial de uma DSL que já seja capaz de modelar diferentes aplicações. A ideia é que a DSL possa evoluir pela contínua reaplicação do processo tendo como entrada aplicações diferentes para produzir novos componentes.

Uma limitação do trabalho está na falta de estudos de casos ou provas de conceito envolvendo aplicações robóticas reais ou pelo menos mais complexas, o que poderia ter resultado em diretrizes mais completas. Isso se deu pela impossibilidade de se utilizar equipamentos robóticos destinados a aplicações reais e, consequentemente, de obter ou implementar tais aplicações. Outras limitações, e que caracterizam também como trabalhos futuros, são a falta de diretrizes específicas a esse domínio nas demais atividades do processo, i.e. implementação do gerador de código e a criação de notações visuais adequadas, e a falta de consideração dos requisitos não-funcionais, os quais são fatores críticos para o sucesso de um sistema embarcado. Ambas as aplicações que foram utilizadas como entrada para criar as DSLs 1 e 2 são feitas utilizando linguagens orientadas a objetos (Java e C++, respectivamente). Muitas aplicações robóticas utilizam outras linguagens de programação, como C, Haskell, e até mesmo Assembly. Por fim, não há uma prova de conceito que suporte a capacidade de evolução da DSL pela reaplicação do processo.

Com a crescente complexidade das aplicações robóticas, faz-se necessário o desenvolvi-

mento de técnicas de Engenharia de Software e ferramentas para melhorar a sua qualidade. Espera-se que este trabalho contribua para essa área e forneça insumos que permitam a evolução das técnicas para lidar com o domínio da robótica.

REFERÊNCIAS

ARKIN, R. *Behavior-based robotics*. Cambridge, MA: MIT Press, 1998. (Intelligent robots and autonomous agents).

ARNOLD, G. V. *Automatization of the Code Generation for Different Industrial Robots*. Tese (Doutorado) — Universidade do Minho, 2007.

BALASUBRAMANIAN, K.; BALASUBRAMANIAN, J.; PARSONS, J.; GOKHALE, A.; SCHMIDT, D. C. A platform-independent component modeling language for distributed real-time and embedded systems. *Journal of Computer and System Sciences*, v. 73, n. 2, p. 171 – 185, 2007. ISSN 0022-0000. Special Issue: Real-time and Embedded Systems.

BRAITENBERG, V. *Vehicles - Experiments in Synthetic Psychology*. Cambridge, MA: The MIT Press, 1984.

BRÄUNL, T. *Embedded Robotics: mobile robot design and applications with embedded systems*. 3rd. ed. [S.l.]: Springer, 2006.

BROOKS, R. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, v. 2, n. 1, p. 14 – 23, mar. 1986. ISSN 0882-4967.

BROOKS, R. *The behavior language; user's guide*. [S.l.]: Citeseer, 1990.

BUFFENBARGER, J.; GRUELL, K. A language for software subsystem composition. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9 - Volume 9*. Washington, DC, USA: IEEE Computer Society, 2001. p. 9072–. ISBN 0-7695-0981-9.

BUNGE, M. *Treatise on basic philosophy*. [S.l.]: D Reidel Pub Co, 1983. ISBN 9027715114.

DEURSEN, A. van; KLINT, P.; VISSER, J. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, p. 26–36, June 2000. ISSN 0362-1340.

DURELLI, R.; CONRADO, D. B. F.; RAMOS, R. A.; PASTOR, O. L.; CAMARGO, V. V.; PENTEADO, R. A. D. Identifying features for ground vehicles software product lines by means of annotated models. In: *Model Based Architecting and Construction of Embedded Systems*. [S.l.: s.n.], 2010. p. 119–123.

DVICE. *MSR-H01-hexapod-Matt-Denton.jpg*. 2011. `http://dvice.com/pics/MSR-H01-hexapod-Matt-Denton.jpg`.

EVERMANN, J.; WAND, Y. Toward formalizing domain modeling semantics in language syntax. *IEEE Transactions on Software Engineering*, IEEE Computer Society, v. 31, n. 1, p. 21–37, 2005. ISSN 0098-5589.

FALBO, R. d. A.; GUIZZARDI, G.; DUARTE, K. C. An ontological approach to domain engineering. In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. New York, NY, USA: ACM, 2002. (SEKE '02), p. 351–358. ISBN 1-58113-556-4.

FIRBY, R. J. Task networks for controlling continuous processes. In: INTERNATIONAL CONFERENCE ON AI PLANNING SYSTEMS, 2., 1994, Menlo Park, CA. *Proceedings...* [S.l.], 1994. p. 49–54.

FOX, D. Kld-sampling: Adaptive particle filters. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 14., 2001. *Proceedings...* [S.l.], 2001. p. 713–720.

FRAKES, W.; PRIETO-;DIAZ, R.; FOX, C. Dare: Domain analysis and reuse environment. *Annals of Software Engineering*, Springer Netherlands, v. 5, p. 125–141, 1998. ISSN 1022-7091. 10.1023/A:1018972323770.

GRAPHICAL Modeling Framework. 2011. `http://www.eclipse.org/modeling/gmp/`. Último acesso em: 17/02/2011.

GROOVER, M. P. *Automatization, Production, Systems, and Computer-Integrated Manufacturing*. [S.l.]: Prentice Hall, 1987.

GÜNTHER, S.; HAUPT, M.; SPLIETH, M. Agile engineering of internal domain-specific languages with dynamic programming languages. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ADVANCES, 5., 2010, Nice, França. *Proceedings...* [S.l.], 2010. p. 162–168.

HAMMOND, K.; MICHAELSON, G. Hume: A domain-specific language for real-time embedded systems. In: *Generative Programming and Component Engineering*. [S.l.]: Springer Berlin / Heidelberg, 2003.

HAMMOND, K.; MICHAELSON, G. Hume: A domain-specific language for real-time embedded systems. In: PFENNING, F.; SMARAGDAKIS, Y. (Ed.). *Generative Programming and Component Engineering*. [S.l.]: Springer Berlin / Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2830). p. 37–56.

HEATH, S. *Embedded systems design*. [S.l.]: Newnes, 2003. (EDN series for design engineers). ISBN 9780750655460.

KELLY, S.; TOLVANEN, J. *Domain-Specific Modeling*. [S.l.]: IEEE Computer Society, 2008.

KELLY, S.; TOLVANEN, J. *Domain-specific modeling: enabling full code generation*. Hoboken, New Jersey: Wiley-IEEE Computer Society Pr, 2008. ISBN 0470036664.

KLEPPE, A.; WARMER, J.; BAST, W. *MDA explained: the model driven architecture: practice and promise*. Boston, MA: Addison-Wesley Professional, 2003. ISBN 032119442X.

KNUTH, K. *Controling Lego NXT Robots with Matlab*. 2011. http://www.huginn.com/knuth/blog/2007/03/30/controlling-lego-nxt-robots-with-matlab/. Último acesso em: 07/03/2001.

KONOLIGE, K. *Colbert: A language for reactive control in sapphira*. [S.l.], jun. 1997.

LATOMBE, J.-C. *Robot motion planning*. [S.l.]: Kluwer Academic Publishers, 1991.

LEGO. *Lego.com MINDSTORMS*. 2011. http://mindstorms.lego.com/. Último acesso em: 07/03/2011.

LEJOS. *Behavior Programming*. 2011. http://lejos.sourceforge.net/nxt/nxj/tutorial/Behaviors/BehaviorProgramming.htm. Último acesso em: 08/03/2011.

LEJOS. *LeJOS, Java for Lego Mindstorms*. 2011. http://lejos.sourceforge.net/. Último acesso em: 07/03/2011.

LIGGESMEYER, P.; TRAPP, M. Trends in embedded software engineering. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 26, p. 19–25, 2009. ISSN 0740-7459.

MDA. 2011. http://www.omg.org/mda/. Último acesso em: 18/02/2011.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 37, p. 316–344, December 2005. ISSN 0360-0300.

METACASE – Domain-Specific Modeling with MetaEdit+. 2011. http://www.metacase.com/. Último acesso em: 17/02/2011.

MICROSOFT. *Microsoft .NET Framework*. 2011. http://www.microsoft.com/net/. Último acesso em: 18/02/2011.

MICROSOFT. *Microsoft Robotics*. 2011. http://msdn.microsoft.com/en-us/robotics/. Último acesso em: 07/03/2011.

MOODY, D. Complexity effects on end user understanding of data models: an experimental comparison of large data model representation methods. In: EUROPEAN CONFERENCE ON INFORMATION SYSTEMS (ECIS 2002), 10., 2002, Gdansk. *Proceedings...* [S.l.], 2002. p. 482–496.

MOODY, D. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, v. 35, n. 6, p. 756 –779, nov.-dec. 2009.

NI, N. I. *NI LabVIEW - Improving the productivity of Engineers and Scientists*. 2011. http://www.ni.com/labview/. Último acesso em: 07/03/2011.

NORDBOTTEN, J. C.; CROSBY, M. E. The effect of graphic style on data model interpretation. *Inf. Syst. J.*, v. 9, n. 2, p. 139–156, 1999.

OMG. *OMG CORBA Website*. 2011. http://www.corba.org/. Último acesso em: 18/02/2011.

OMG – Object Management Group – UML. 2011. `http://www.uml.org/`. Último acesso em: 14/02/2011.

OMG MetaObject Facility (MOF). 2011. `http://www.omg.org/mof/`. Último acesso em: 18/02/2011.

ORACLE. *JavaServer Faces Technology*. 2011. `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html`. Último acesso em: 18/02/2011.

PETERSON, J.; HAGER, G. Monadic robotics. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, p. 95–108, December 1999. ISSN 0362-1340.

PLAYER/STAGE/GAZEBO Project. 2011. `http://playerstage.sourceforge.net/`. Último acesso em: 15/02/2011.

ROBERT, S.; GÉRARD, S.; TERRIER, F.; LAGARDE, F. A lightweight approach for domain-specific modeling languages design. In: EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, 35., 2009, Patras, Grécia. *Proceedings...* [S.l.], 2009. p. 155–161.

ROBOTSHOP. *dfrobotshop-rover-lights-B.jpg*. 2011. `http://www.robotshop.com/Images/xbig/en/dfrobotshop-rover-lights-B.jpg`.

ROPER, M. D.; OLSSON, R. A. Developing embedded multi-threaded applications with catapults, a domain-specific language for generating thread schedulers. In: *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2005. (CASES '05), p. 295–303. ISBN 1-59593-149-X.

SIEGWART, R.; NOURBAKHSH, I. *Introduction to Autonomous Mobile Robots*. Cambridge, MA: The MIT Press, 2004.

STREMBECK, M.; ZDUN, U. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, v. 39, p. 1253–1292, 2009.

ULRICH, I.; BORENSTEIN, J. Vfh+: Reliable obstacle avoidance for fast mobile robots. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1998, Leuven, Belgium. *Proceedings...* [S.l.], 1998. v. 2, p. 1572–1577.

VISUAL Studio Visualization and Modeling SDK (was DSL SDK). 2011. `http://code.msdn.microsoft.com/vsvmsdk`. Último acesso em: 17/02/2011.

WEBER, R.; ZHANG, Y. An analytical evaluation of niam's grammar for conceptual schema design. *Information Systems Journal*, v. 6, n. 2, p. 147–170, 1996.

WOLF, W.; FREY, E. Tutorial on embedded system design. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN ON VLSI IN COMPUTER & PROCESSORS, 1991, Washington, DC. *Proceedings...* Washington, DC, USA: IEEE Computer Society, 1992. (ICCD '92), p. 18–21. ISBN 0-8186-3110-4.

WOLF, W. H. *Computers as Components: Principles of Embedded Computing System Design*. 2nd. ed. Amsterdam: Morgan Kaufmann Publishers Inc., 2008.

XMI. 2011. `http://www.omg.org/spec/XMI/`. Último acesso em: 18/02/2011.

Apendice A

CÓDIGO-FONTE DO ROBÔ MONITORADOR DE PORTAS

A.1 Arquivo Hallmonitor.java: o programa principal

```
1  package br.ufscar.dc.db.securityapp.hallmonitor;
2
3  import lejos.nxt.*;
4  import lejos.robotics.subsumption.*;
5  import br.ufscar.dc.db.securityapp.hallmonitor.behavior.*;
6
7  public class Hallmonitor {
8
9    public static void main(String[] args) {
10     final Behavior driveForward = new DriveForward();
11     final Behavior doorOpened = new DoorOpened(SensorPort.S1);
12     final Behavior turnBack = new TurnBack(SensorPort.S3, 2.5f, 11.5f);
13
14     final Arbitrator arby = new Arbitrator(
15     new Behavior[] {driveForward, turnBack, doorOpened});
16     arby.start();
17   }
18 }
```

A.2 Arquivo DriveForward.java: locomoção do robô

```
1  package br.ufscar.dc.db.securityapp.hallmonitor.behavior;
2
3  import lejos.nxt.*;
4  import lejos.robotics.subsumption.*;
5
6  public class DriveForward implements Behavior {
7
8          private boolean suppressed = false;
9
10         public boolean takeControl() {
11                 return true;
12         }
13
14         public void suppress() {
15                 suppressed = true;
16         }
17
18         public void action() {
19                 suppressed = false;
20
21                 Motor.B.forward();
22                 Motor.C.forward();
23
24                 while (!suppressed) {
25                         Thread.yield();
26                 }
27
28                 Motor.B.stop();
29                 Motor.C.stop();
30         }
31 }
```

A.3 Arquivo DoorOpened.java: detector de portas abertas

```
1  package br.ufscar.dc.db.securityapp.hallmonitor.behavior;
```

```
2
3  import lejos.nxt.*;
4  import lejos.robotics.subsumption.*;
5
6  import br.ufscar.dc.db.securityapp.hallmonitor.*;
7  import static br.ufscar.dc.db.securityapp.hallmonitor.util.Util.*;
8  import static java.lang.Math.*;
9
10 public class DoorOpened implements Behavior {
11
12   private static final int UNKNOWN_DISTANCE = 300;
13
14   private UltrasonicSensor sensor;
15   private int firstDistance = UNKNOWN_DISTANCE;
16   private int priorDistance = UNKNOWN_DISTANCE;
17   private boolean doorDetected = false;
18   private int tamanhoBatente = 13;
19   private boolean doorOpenedDetected = false;
20   private int lastDistanceSinceDoorDetected = UNKNOWN_DISTANCE;
21
22   private boolean suppressed = false;
23
24   public DoorOpened(SensorPort sensorPort) {
25       sensor = new UltrasonicSensor(sensorPort);
26   }
27
28   public boolean takeControl() {
29       final int distance = sensor.getDistance();
30       System.out.println("distance: " + distance);
31
32       // descartar erros de leitura.
33       if (distance == 255)
34           return false;
35
36       // pegar a primeira distancia, da qual o robo ira se pautar.
37       // assume-se que a primeira distancia medida eh a distancia
38       // entre o robo e a parede e que o robo irá andar em linha
39       // reta ao longo do corredor.
```

```
40        if (firstDistance == UNKNOWN_DISTANCE) {
41            firstDistance = distance;
42            return false;
43        }
44
45        if (priorDistance != UNKNOWN_DISTANCE) {
46          // fazemos com que o robo continue indo em linha reta
47          // calculando a diferença entre a distancia anterior
48          // e a distancia atual, e determinando a velocidade
49          // da roda direita para que ela corrija o caminho.
50          // por exemplo, se a diferença for positiva, significa
51          // que o robo está se aproximando da parede e, portanto,
52          // é preciso aumentar a velocidade da roda direita para
53          // que ele comece a se afastar da parede.
54          int instantDiff = abs(priorDistance - distance);
55          boolean firstTimeDoorDetected = false;
56
57          if (isBetween(instantDiff,tamanhoBatente-1,tamanhoBatente+1)){
58            firstTimeDoorDetected = true;
59            lastDistanceSinceDoorDetected = priorDistance;
60            System.out.println("Last Distance Since Door Detected: "
61                      + lastDistanceSinceDoorDetected);
62            doorDetected = !doorDetected;
63            if (doorDetected)
64              System.out.println("Door Detected");
65            else
66              System.out.println("End of door");
67          }
68
69          if (doorOpenedDetected &&
70                    isBetween(distance, lastDistanceSinceDoorDetected-1,
71                              lastDistanceSinceDoorDetected+1)) {
72            doorOpenedDetected = false;
73            System.out.println("End of Opened Door");
74          } else if (!firstTimeDoorDetected && (instantDiff >= 3)) {
75            doorOpenedDetected = true;
76            System.out.println("** Door Opened");
77            Sound.beep();
```

```
78          } else if (!doorOpenedDetected) {
79            int diff = firstDistance - distance;
80
81            if (doorDetected) {
82              System.out.println("Retirando o tamanho do batente");
83              diff -= tamanhoBatente;
84            }
85
86            float speed = 360f;
87            if (diff != 0)
88              speed += diff /*/ 7f*/;
89            System.out.println("Motor speed: " + speed);
90            Motor.C.setSpeed(speed);
91          }
92        }
93
94        priorDistance = distance;
95        return false;
96    }
97
98    public void suppress() {
99        suppressed = true;
100   }
101
102   public void action() {
103   }
104 }
```

A.4 Arquivo TurnBack.java: detecção do fim do corredor

```
1  package br.ufscar.dc.db.securityapp.hallmonitor.behavior;
2
3  import lejos.nxt.*;
4  import lejos.robotics.subsumption.*;
5
6  /**
7   * Este comportamento atua quando o sensor de toque e' pressionado.
```

```
8   * Dai ele da meia volta.
9   */
10  public class TurnBack implements Behavior {
11
12      private boolean suppressed = false;
13      private TouchSensor sensor;
14      private double wheelDiameter;
15      private double trackWidth;
16
17      public TurnBack(SensorPort touchSensorPort,
18      double wheelDiameter, double trackWidth) {
19          sensor = new TouchSensor(touchSensorPort);
20          this.wheelDiameter = wheelDiameter;
21          this.trackWidth = trackWidth;
22      }
23
24      public boolean takeControl() {
25          return sensor.isPressed();
26      }
27
28      public void suppress() {
29          suppressed = true;
30      }
31
32      public void action() {
33          suppressed = false;
34
35          // eh necessario que o robo volte para tras distancia
36          // suficiente para que ele possa girar em torno de
37          // si mesmo meia volta sem bater na parede.
38          Motor.A.rotate(-360, true);
39          Motor.C.rotate(-360, true);
40
41          while (Motor.A.isMoving())
42              if (!suppressed)
43                  Thread.yield();
44              else
45                  return;
```

```
46
47          // agora, eh necessario fazer a meia volta. O robo precisa
48          // girar uma roda para frente e outra para tras para que
49          // ele vire em torno de si mesmo. a quantidade de voltas
50          // que a roda deve dar deve ser suficiente para que ele
51          // faca a meia volta. Como calcular isso? A maneira mais
52          // simples eh ter como entrada o diametro das rodas
53          // e a distancia entre elas.
54          final int angle = 180;
55          final int fakeAngle = angle * (int)(trackWidth/wheelDiameter);
56          Motor.A.rotate(fakeAngle, true);
57          Motor.C.rotate(-fakeAngle, true);
58
59          while (Motor.A.isMoving())
60              if (!suppressed)
61                  Thread.yield();
62      }
63  }
```

A.5 Arquivo Util.java: utilitário

```
1   package br.ufscar.dc.db.securityapp.hallmonitor.util;
2
3   public class Util {
4
5           public static boolean isBetween(int x, int b1, int b2) {
6                   return (b1 <= x && x <= b2);
7           }
8   }
```

Apendice B

CÓDIGO-FONTE DO ROBÔ ENTREGADOR DE MATERIAIS

B.1 Arquivo deliverer.cc: código de controle do robô

```
#include <iostream>
#include <libplayerc++/playerc++.h>
#include <boost/array.hpp>
#include "roomreceiver.h"

typedef struct {
    double x;
    double y;
    double aw;
} objetivo_pose_t;

boost::array<objetivo_pose_t, 2> rooms;

int
main(int argc, char *argv[])
{
    // inicializacao dos devices conigurados
    // no arquivo cfg
    PlayerCc::PlayerClient client("localhost");
    PlayerCc::Position2dProxy p2d(&client, 0);
    PlayerCc::LaserProxy lp(&client, 0);
```

```
22      PlayerCc::SpeechProxy sp(&client, 0);
23      PlayerCc::MapProxy mp(&client, 0);
24      PlayerCc::Position2dProxy p2d1(&client, 1);
25      PlayerCc::Position2dProxy p2d2(&client, 2);
26      PlayerCc::PlannerProxy pp(&client, 0);
27
28      // dados das salas
29      rooms[0].x = 2;
30      rooms[0].y = -4;
31      rooms[0].aw = 0;
32
33      rooms[1].x = 3.5;
34      rooms[1].y = -3;
35      rooms[1].aw = 1;
36
37      // o listener que vai processar a entrada do usuario
38      // quando ele informar a sala em que o robo deve ir
39      class : public RoomListener
40      {
41          public:
42          PlayerCc::PlannerProxy *pp;
43          PlayerCc::SpeechProxy *sp;
44          virtual void roomReceived(int room)
45          {
46            try
47            {
48              objetivo_pose_t pose = rooms.at(room);
49              pp->SetGoalPose(pose.x, pose.y, pose.aw);
50              pp->RequestWaypoints();
51              pp->SetEnable(true);
52            }
53            catch (std::range_error& e)
54            {
55              sp->Say("I don't know room number " + room);
56            }
57          }
58      } listener;
59      listener.pp = &pp;
```

```
60      listener.sp = &sp;
61
62      try
63      {
64          // o robo abre um server socket UDP e fica esperando
65          // o usuario enviar para ele a sala em que ele deve ir
66          // atraves do computador. O computador sabe o endereco
67          // IP e a porta do robo.
68          RoomReceiver receiver(8080);
69          receiver.setRoomListener(&listener);
70          receiver.run();
71      }
72      catch (PlayerCc::PlayerError pe) {
73          std::cout << "Error:" << pe << std::endl;
74      }
75  }
```

B.2 Arquivo roomreceiver.h: implementação da comunicação via proxy

```
1   #ifndef _ROOMRECEIVER_H
2   #define _ROOMRECEIVER_H
3
4   #include <boost/array.hpp>
5   #include <boost/asio.hpp>
6
7   using boost::asio::ip::udp;
8
9   class RoomListener
10  {
11    public:
12    virtual ~RoomListener() {}
13    virtual void roomReceived(int room) = 0;
14  };
15
16  class RoomReceiver
```

```
{
  private:
  RoomListener *listener;
  int port;
  public:
  RoomReceiver(int _port) : port(_port)
  {}
  void setRoomListener(RoomListener *l)
  {
        listener = l;
  }

  void run()
  {
    boost::asio::io_service io_service;
    udp::socket socket(io_service, udp::endpoint(udp::v4(), port));
    for (;;)
    {
      boost::array<int, 1> recv_buf;
      udp::endpoint remote_endpoint;
      boost::system::error_code error;
      socket.receive_from(boost::asio::buffer(recv_buf),
        remote_endpoint, 0, error);
      if (error && error != boost::asio::error::message_size)
        throw boost::system::system_error(error);
      int room = *recv_buf.data();
      listener->roomReceived(room);
      // responder
      std::string msg = "OK";
      boost::system::error_code ignored_error;
      socket.send_to(boost::asio::buffer(msg),
        remote_endpoint, 0, ignored_error);
    }
  }
};

class RoomSender
{
```

```
55    private:
56    std::string host, port;
57    public:
58    RoomSender(std::string _host, std::string _port):
59    host(_host), port(_port)
60    {}
61    char * sendRoom(int room)
62    {
63      boost::asio::io_service io_service;
64      udp::resolver resolver(io_service);
65      udp::resolver::query query(host, port);
66      udp::endpoint receiver_endpoint = *resolver.resolve(query);
67      udp::socket socket(io_service);
68      socket.open(udp::v4());
69      boost::array<int, 1> send_buf = {{ room }};
70      socket.send_to(boost::asio::buffer(send_buf),
71        receiver_endpoint);
72      boost::array<char, 128> recv_buf;
73      udp::endpoint sender_endpoint;
74      size_t len = socket.receive_from(
75        boost::asio::buffer(recv_buf), sender_endpoint);
76      return recv_buf.data();
77    }
78  };
79
80  #endif
```

B.3 Arquivo autolab2.world: definição do mundo virtual para o simulador

```
1  # the resolution of Stage's raytrace model in meters
2  resolution 0.02
3
4  interval_sim 100  # milliseconds per update step
5  interval_real 0 # real-time milliseconds per update step
6
```

```
include "pioneer.inc"
include "map.inc"
include "sick.inc"

paused 1

# configure the GUI window
window
(
  size [ 678.000 730.000 ]
  center [0.122 -0.386]
  scale 31.082
)

# load an environment bitmap
floorplan
(
  bitmap "bitmaps/autolab.png"
  size [20.000 20.000 0.500]
  boundary 1
  name "lab"
)

pioneer2dx
(
  # can refer to the robot by this name
  name "r0"
  pose [ 2 -6 0 45 ]

  sicklaser(
     # ctrl "lasernoise"
  )

  # report error-free position in world coordinates
  localization "gps"
  localization_origin [ 0 0 0 0 ]
)
```

B.4 Arquivo autolab2.cfg: configurações dos dispositivos do robô

```
driver
(
  name "stage"
  provides ["simulation:0"]
  plugin "stageplugin"
  #plugin "libstage"

  # load the named file into the simulator
  worldfile "autolab2.world"
)


driver
(
  name "stage"
  provides ["odometry::position2d:0" "laser:0" "speech:0"]
  model "r0"
)

# Load the map for localization and planning from the same image file,
# and specify the correct resolution (a 500x500 pixel map at 16m x 16m
# is 0.032 m / pixel resolution).
driver
(
  name "mapfile"
  filename "bitmaps/autolab.png"
  resolution 0.032
  provides ["map:0"]
)

driver
(
  name "amcl"
  provides ["position2d:2"]
  requires ["odometry::position2d:1" "laser:0" "laser:::map:0"]
```

```
36    init_pose [2 -6 0]
37    update_thresh [0.1 5]
38  )
39
40  driver
41  (
42    name "vfh"
43    provides ["position2d:1"]
44    requires ["position2d:0" "laser:0"]
45    safety_dist 0.1
46    distance_epsilon 0.3
47    angle_epsilon 5
48  )
49
50  driver
51  (
52    name "wavefront"
53    provides ["planner:0"]
54    requires ["output:::position2d:1" "input:::position2d:2" "map:0"]
55    safety_dist 0.15
56    distance_epsilon 0.5
57    angle_epsilon 10
58  )
```