Geração de Código para Robôs implementados com ROS a partir de modelos UML/MARTE

Marco Aurélio Wehrmeister
Universidade Tecnológica Federal do Paraná (UTFPR)
Av. Sete de Setembro, 3165 - CEP 80230-901 - Curitiba - PR - Brasil
wehrmeister@utfpr.edu.br
http://orcid.org/0000-0002-1415-5527

***Abstract*— Este trabalho apresenta uma abordagem para a geração de código fonte a partir de modelos UML anotados com o perfil MARTE para sistemas embarcados usados em robôs. A abordagem AMoDE-RT foi estendida para suportar a geração de código C++ que utiliza a semântica e as bibliotecas disponíveis no *Robot Operating System* (ROS). O ROS é um framework amplamente utilizado no projeto e implementação do software para robôs tanto no meio industrial como no acadêmico. O objetivo principal deste trabalho é a aplicação de técnicas de engenharia guiada por modelos (MDE) (suportadas na AMoDE-RT) no projeto de robôs visando a prototipação rápida de robôs através de simuladores e também a implementação real deles. O trabalho foi validado através de um estudo de caso que mostra a viabilidade da proposta na implementação de um robô simples. Os resultados obtidos fornecem indicativos que a abordagem pode ser aplicada em sistemas mais complexos que envolvem múltiplos robôs que interagem para realizar tarefas em ambientes industriais.**

Keywords — geração de código, engenharia guiada por modelos, UML, MARTE, Robot Operating Systems (ROS)

I. INTRODUÇÃO

Os robôs são sistemas complexos compostos por componentes de hardware sofisticados e heterogêneos [13][24]. Uma arquitetura complexa de software coordena a execução de algoritmos que interagem e controlam estes componentes [21]. Especialistas de várias áreas estão envolvidos no projeto de um sistema robótico [20][17], e.g. engenheiros mecânicos, eletrônicos, de controle, cientistas da computação, etc. Durante o projeto do robô, os especialistas precisam interagir entre si, buscando desenvolver e integrar os pedaços do software que resolvem problemas específicos do seu domínio. Entretanto, não existe nenhum apoio real para essa cooperação [21][22][2].

A complexidade dos sistemas robóticos cresce drasticamente ao adicionar funcionalidades e trabalhar fora dos componentes preexistentes. Muitas vezes os desenvolvedores do software do sistema robótico não estão preparados para esse aumento de complexidade [24][15], especialmente quando eles são especialistas em um domínio específico sem um sólido histórico no desenvolvimento e engenharia de software. Um exemplo típico é dado por especialistas em controle portando seus próprios algoritmos de controle de um ambiente simulado para o robô real. Esses engenheiros precisam não apenas de uma visão geral de arquitetura do sistema (i.e. estrutura e as conexões entre os componentes) como também devem ajudar a projetar o funcionamento interno de cada um desses componentes.

Portanto, é necessário aumentar o nível de abstração usado no projeto do sistema robótico. Abordagens baseadas nas técnicas da engenharia guiada por modelos (do inglês, *Model-Driven Engineering* – MDE) podem auxiliar nas questões mencionadas [1][2][23][20]. A integração automatizada da informação gerada pelas diferentes equipes de engenheiros é um fator chave para o sucesso do projeto. As novas aplicações previstas no contexto da Industria 4.0 irão demandar equipes cada vez mais interdisciplinares e, portanto, a integração dos artefatos produzidos durante o projeto é fundamental.

Além disso, é necessário ter tecnologias de implementação que facilitam a integração dos vários componentes e (sub)sistemas que compõem um sistema robótico. Frameworks como o *Robot Operating System* (ROS) [13][14] fornecem um grande auxílio na implementação do software de um robô ou sistema robótico. Hoje, a principal vantagem do ROS é sua grande comunidade de desenvolvedores e a grande quantidade de componentes de software disponíveis para uso. Um dos princípios do ROS é fornecer uma liberdade ampla para seus desenvolvedores e usuários [21]. No entanto, considerando o crescente aumento da complexidade nos sistemas robóticos, tal liberdade vem acompanhada de uma falta de estrutura no desenvolvimento do software e sua arquitetura. Isso pode dificultar o projeto do sistema robótico como um todo, principalmente do software. De fato, já existiam esforços para adotar uma abordagem mais formal ao desenvolvimento de robôs baseados em ROS [19][18][23][24][22].

Visando abordar os problemas mencionados, este trabalho propõe o uso de uma abordagem baseada na MDE que vem sendo aplicada com sucesso no projeto de sistemas embarcados de tempo-real aplicados em sistemas de automação. A abordagem AMoDE-RT [1][2][3] foi estendida para suportar a geração de código do software embarcado que controla os componentes de um sistema robótico. O código fonte gerado utiliza os conceitos e bibliotecas disponibilizados no ROS para a implementação dos componentes e (sub)sistemas dos robôs que compõem o sistema. Sendo assim, as contribuições deste trabalho são: (i) o mapeamento de elementos disponíveis no modelo UML/MARTE em conceitos e construções do ROS; e (ii) a implementação destes mapeamentos na forma de scripts de geração de código.

O trabalho foi validado através de um estudo de caso que mostrou a viabilidade da proposta. Um robô seguidor de linha especificado em um trabalho anterior [3] foi usado neste estudo de caso. Foi possível gerar uma quantidade razoável de código fonte C++/ROS a partir do modelo UML/MARTE criado em [3]. Apesar de simples, o estudo de caso forneceu indícios sobre a aplicabilidade da abordagem AMoDE-RT no projeto de sistemas robóticos mais complexos.

O restante deste texto foi organizado da seguinte maneira: a Seção II apresenta uma visão geral da abordagem AMoDE-RT e dos elementos que compõem o ROS; a Seção III apresenta a abordagem de geração automática de código C++/ROS a partir de modelos UML/MARTE; a Seção IV apresenta o estudo de caso usado para validar a abordagem proposta; a Seção V apresenta e discute alguns trabalhos relacionados, enquanto a Seção VI finaliza o artigo apresentando as conclusões e os trabalhos futuros.

II. VISÃO GERAL DE CONCEITOS IMPORTANTES

A. AMoDE-RT: Aspect-oriented Model-Driven Engineering for Embedded Real-Time Systems

***Aspect-oriented Model-Driven Engineering for Real-Time systems* (AMoDE-RT)** [1][2][3] é uma abordagem que utiliza técnicas da MDE no projeto de sistemas embarcados de tempo-real. Tal abordagem promove o uso de modelos independentes de plataforma (do inglês, *Platform Independent Model* – PIM) como os artefatos principais produzidos no projeto para especificar os requisitos de sistemas embarcados de tempo-real, incluindo aspectos estruturais, comportamentais, restrições e requisitos não-funcionais. Os PIM são especificados usando a linguagem UML [4] e perfil MARTE [5]. Uma das principais contribuições da AMoDE-RT é a separação das preocupações com o tratamento dos requisitos funcionais e não funcionais. Para isso, conceitos do paradigma de Desenvolvimento de Software Orientado a Aspectos (do inglês, *Aspect-Oriented Software Development* – AOSD) [6] são suportados em modelos UML/MARTE através do ***Distributed Embedded Real-time Aspect Framework* (DERAF)** [7][3]. O DERAF fornece um conjunto de aspectos que definem o tratamento de alguns requisitos não funcionais comumente encontrados no projeto de sistemas embarcados de tempo-real [2][7].

Na AMoDE-RT, a implementação do sistema é obtida automaticamente a partir da informação especificada nos modelos UML/MARTE por meio de transformações de modelo [1][3]. Para isso, propõe-se o uso de um modelo intermediário independente de plataforma chamado ***Distributed Embedded Real-Time Compact Specification* (DERCS)** [8][2]. O modelo DERCS de um sistema embarcado de tempo-real representa requisitos funcionais em termos de conceitos de paradigma orientado a objetos, enquanto requisitos não-funcionais são representados usando conceitos de paradigma orientado a aspectos. A ferramenta ***Generation of Embedded Real-Time Code based on Aspects* (GenERTiCA)** [9] suporta a transformação UML-para-DERCS, assim como a geração automática de código fonte a partir do modelo DERCS. Ela implementa uma abordagem de geração de código baseada em scripts. Cada script é responsável por mapear um elemento de modelo DERCS em construções, serviços e/ou APIs fornecidos por uma determinada plataforma alvo de implementação. Os engenheiros podem especificar scripts de regras de mapeamento para plataformas alvo distintas, incluindo software e hardware, e.g. C/C++ [1], real-time Java [1], web services [11], VHDL [3][12]). Portanto, este trabalho propõe um novo conjunto de regras de mapeamento (i.e. scripts para a geração de código) para C++ com suporte para semântica e as bibliotecas disponíveis no ROS visando a implementação do software embarcado de robôs. A Figura 1 mostra a abordagem de geração de código implementada na ferramenta GenERTiCA, incluindo os scripts que implementam as regras de mapeamento para gerar código C++/ROS.

É importante ressaltar que a GenERTiCA não gera apenas o código fonte, mas também realiza o entrelaçamento de aspectos. Em outras palavras, a GenERTiCA realiza adaptações especificadas pelos aspectos no código gerado, permitindo o uso de conceitos de AOSD, mesmo que a linguagem de implementação de destino não suporte tais conceitos. Para isso, a GenERTiCA utiliza as informações fornecidas pelos aspectos do DERAF, que são especificados de uma forma independente de plataforma nos modelo UML/MARTE. As adaptações de aspectos são implementadas como scripts de regras de mapeamento, permitindo sua portabilidade para diferentes plataformas e/ou aplicativos. Uma adaptação pode ocorrer de duas maneiras: no nível do modelo ou no nível do código. No nível do modelo, as adaptações podem incluir ou alterar elementos no modelo DERCS [9][8] gerados a partir da especificação UML/MARTE. Essas modificações são acessíveis às regras de mapeamento durante a etapa de geração de código. Por outro lado, no nível de código, modificações afetam diretamente o código gerado sem alterar o modelo de entrada. Assim, um trabalho futuro importante é a implementação dos aspectos do DERAF usando as preocupações transversais (em inglês, *crosstutting concerns*) implementadas no ROS. Mais detalhes sobre o AMoDE-RT, DERAF e GenERTiCA podem ser encontrados em [1], [2], [3], [8] e [9].

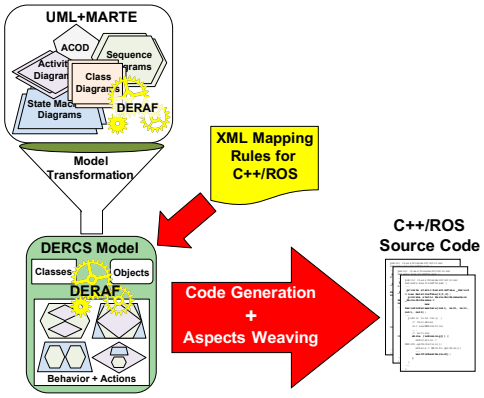

Fig. 1. Processo de geração de código fonte implementado na GenERTiCA

B. ROS: Robot Operating System

O ROS é um framework de código aberto para a construção de robôs [13][14]. O ROS fornece alguns serviços de um sistema operacional, e.g. abstração de hardware, controle de dispositivos com baixo nível de abstração, implementação de funcionalidades e serviços comuns a vários processos, troca de mensagens entre processos, e gerenciamento de pacotes. Além disso, ele fornece ferramentas e bibliotecas para obter, criar, gravar e executar código em vários computadores em um ambiente distribuído. O ROS é implementado para rodar em sistemas operacionais baseados em Unix e outros de forma limitada e restrita. Seu principal foco é na padronização de mensagens e a troca delas entre os vários nós ROS. Tal padronização permite o compartilhamento de soluções facilitando o reuso.

O modelo de execução do ROS é uma rede *peer-to-peer* de processos que são fracamente acoplados por usar a infraestrutura de comunicação ROS. Tais processos podem ser distribuídos em máquinas distintas conectadas em uma infraestrutura de comunicação, incluindo a Internet. O ROS implementa vários estilos diferentes de comunicação: (a) comunicação síncrona no estilo chamada remota de procedimento (em inglês, *Remote Procedure Call* - RPC) sobre serviços; (b) Fluxo assíncrono de dados sobre tópicos; e (c) armazenamento e compartilhamento de dados em um servidor de parâmetros.

O ***grafo de computação*** é a rede peer-to-peer de processos implementada no ROS [14]. Os ***nós ROS*** são nós deste grafo e representam os processos que realizam alguma computação e.g. interface com sensores e atuadores, comunicação entre dispositivos, navegação e mapeamento, controladores, etc. Um robô é constituído por um ou vários nós ROS que são

implementados usando alguma biblioteca de cliente ROS, e.g. C++, Python e Lisp [13]. Um dos nós do grafo de computação representa o **mestre ROS**. Ele fornece o serviço de registro de nomes dos nós ROS, seus serviços e tópicos, além de um serviço de busca deles no grafo de computação. O mestre ROS permite que os nós ROS encontrem-se uns aos outros, e consequentemente, realizem a comunicação direta entre si e o uso direto dos serviços fornecidos por cada nó ROS. Quando um nó ROS se conecta no grafo de computação, ele se registrar no mestre ROS e fornecer informações sobre os tópicos e serviços fornecidos por ele.

Após o processo de busca no mestre ROS, os nós ROS se conectam diretamente. Como mencionado, a comunicação entre eles pode acontecer de duas formas: síncrona e assíncrona. Na comunicação síncrona, um nó ROS solicita ***serviços*** de outro nó de forma similar ao RPC. Dessa forma o nó que faz a chamada fica bloqueado esperando até que o nó remoto forneça uma resposta após o término da execução completa do serviço solicitado. Já a comunicação assíncrona segue o paradigma *publisher-subscriber* [10] que é implementado através de ***tópicos ROS***. Um nó ROS publica ***mensagens*** em um ou mais tópicos. Por sua vez, os nós ROS assinam tópicos e são notificados da chegada da nova mensagem. Uma ***mensagem ROS*** pode ser vista como uma estrutura que agrupa dados simples (e.g. inteiros) ou complexos (e.g. pose do robô composta pela sua posição no espaço tridimensional e o ângulo de visada). Tal abordagem permite a comunicação muitos-para-muitos de forma desacoplada, i.e. os nós ROS não precisam conhecer uns aos outros para trocar dados através das mensagens nos tópicos. ROS suporta vários protocolos de comunicação, porém os mais usados são o TCP/IP [13]. A Figura 2 mostra um exemplo do grafo de computação.

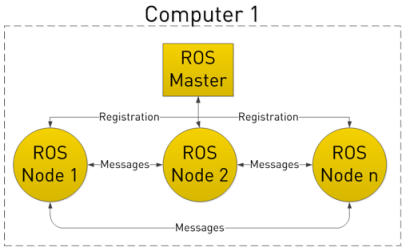

Fig. 2. Grafo de computação do ROS [14]

Por fim, é importante destacar que o mestre ROS pode armazenar dados centralizados que estão disponíveis a todos os nós ROS através de um ***servidor de parâmetros***. Adicionalmente, o ROS fornece um meio de salvar e refazer o histórico das mensagens enviadas no grafo de computação através de ***bags***. Um *bag p*ode ser visto como um log de mensagens e dados trocados entre os nós ROS e é um recurso bastante usado durante o desenvolvimento do robô para testar, verificar e validar o sistema.

III. Geração Automática de Código para Robôs Usando ROS

Este trabalho propõe que a geração automática do código fonte para o software dos sistemas embarcados que controlam o robô. A implementação deste software utiliza o ROS e se baseia na semântica de comunicação *publisher-subscriber* e chamada de serviços.

O código fonte do software é gerado automaticamente pela ferramenta GenERTiCA [9] que recebe como entrada o modelo intermediário DERCS que representa os requisitos funcionais e não-funcionais dos vários sistemas embarcados que compõem os robôs que fazem parte do sistema robótico como um todo. A ferramenta GenERTiCA analisa os objetos nesta descrição intermediária e aplica as regras de mapeamento que foram desenvolvidas anteriormente e estão armazenadas em um repositório de regras de mapeamento.

As regras de mapeamento são descritas em arquivos XML [2][3]. O formato XML foi escolhido devido a sua estrutura em árvore que melhora a reutilização de regras de mapeamento desenvolvidas anteriormente, como por exemplo, as regras de mapeamento para C++ que foram criadas em [1] e reusadas e adaptadas neste trabalho. Ao usar arquivos XML, o engenheiro pode criar um repositório de regras de mapeamento que foram desenvolvidas e validadas anteriormente para que possam ser reutilizadas.

Os nós-folha da árvore de regras de mapeamento contêm scripts que permitem a geração de código a partir do modelo DERCS que representa os elementos do sistema seguindo a semântica do paradigma orientado a objetos. Dependendo da classificação do objeto (por exemplo, software ou hardware, local ou remoto, objeto ativo ou passivo), um script específico é selecionado (da árvore de regras de mapeamento) e executado, resultando em um fragmento de código fonte específico que pode ser C/C++ [1], real-time Java [1], VHDL [3][12]), SystemC, etc. A linguagem de script tem acesso total as informações sobre objetos do sistema, i.e. sua estrutura e seu comportamento. Portanto é possível construir regras de mapeamento especializadas e complexas como as criadas no contexto deste trabalho.

Inicialmente é necessário estabelecer o mapeamento dos objetos que estão contidos no modelo UML/MARTE com os elementos estruturais da plataforma alvo. No caso do ROS, os elementos estruturais são os nós ROS que são mapeados em processos no sistema operacional. Assim, um nó ROS pode conter mais de um objeto. No entanto, considerando a natureza distribuída do ROS, este trabalho propõe o mapeamento 1-para-1 entre objetos no modelo UML/MARTE e nós ROS, i.e. para cada objeto, um novo nó ROS é criado. Isso permite que os objetos que compõem o sistema robótico (i.e. os nós ROS) possam ser executados de forma distribuída.

Além do código referente as classes, atributos e métodos, é necessário incluir a função *main()* com um código de inicialização do ROS para manter o nó em execução. Adicionalmente, caso o objeto seja ativo, i.e. sua classe foi anotada com o estereótipo *<<SchedulableResource>>* do perfil MARTE, o método que representa o comportamento ativo deve ser invocado logo após a inicialização do ROS. Geralmente, um objeto ativo tem seu comportamento executado de forma cíclica (i.e. tarefa periódica) com frequência de execução controlada. Tal informação é obtida através do estereótipo *<<TimedEvent>>* do perfil MARTE é usado no modelo UML para anotar a ativação do comportamento do objeto ativo. O código que implementa a execução cíclica e também o controle da frequência de execução é gerado a partir do aspecto *PeriodicTiming* do DERAF. Este trabalho propõe um mapeamento para este aspecto em construções disponíveis na biblioteca do ROS. Além disso, no caso dos objetos passivos, i.e. aqueles que fornecem serviços em resposta a chamada de outros objetos (ativos ou passivos), é necessário incluir um *loop infinito* para permitir que nó ROS esteja executando e possa responder adequadamente as chamadas de serviços ou a novas

mensagens enviadas nos tópicos ROS. A Figura 3 mostra o script de geração de código para um nó ROS. O texto na cor preta mostra do script que gera a definição da class, enquanto o texto em vermelho mostra a geração de código de inicialização do ROS. Um exemplo concreto da geração de código de um nó ROS é apresentado na próxima seção.

```
#set( $uh = '_h' )
#set( $OBJECT_NAME = "$Class.Name_$Object.Name" )

#ifndef $Class.Name$uh
#define $Class.Name$uh

## generating the class definition
class $Class.Name
#if ($Class.SuperClass)
   : public $Class.SuperClass.Name
#end
$Options.BlockStart
protected:
$CodeGenerator.getAttributesDeclaration(1)

$CodeGenerator.getMessagesDeclaration(1)
$Options.BlockEnd
;

## generating the main function
int main(int argc, char **argv) {
  ros::init(argc, argv, "$OBJECT_NAME");
  ros::NodeHandle $OBJECT_NAME;

  ## code for the provided services (servers) (see fig. 4)
  ## code for the requested services (clients) (see fig. 5)
  ## code for the published/subscribed topics (see fig. 6)

  #if ($Object.isActive)
     $OBJECT_NAME."$Object.MainBehavior.Name"()";
  #else
     ros::spin();
  #end

  return 0;
}

#endif
```

Fig. 3. Script para a geração de código de um nó ROS: definição da classe, função *main()* e inicialização do ROS

Uma parte importante da especificação das regra de mapeamento é a especificação da troca de mensagens entre os objetos que compõem o sistema. No modelo DERCS, os objetos trocam mensagens com outros objetos local (i.e. que residem e executam na mesma unidade de processamento, seja ela uma CPU, GPU ou área em FPGA) ou objetos remotos (i.e. que residem e executam outra unidade de processamento que está localizada em outro local físico mas está interconectada através de uma infraestrutura de comunicação). O mesmo é verdadeiro para a comunicação entre objetos locais ou remotos que são implementados em tecnologias distintas como software e hardware. Estas formas de comunicação são implementadas com scripts específicos e diferentes. A ferramenta GenERTiCA gera o código apropriado para cada tipo de mensagem a ser enviada (e.g. local/local, local/remoto, software/hardware, etc.), executando o script associado a cada nó filho do nó *MessageExchange*. Por exemplo, considerando um objeto implementado em software precisa enviar uma mensagem para um objeto remoto também implementado em software, o script que implementa essa comunicação pode descrever qualquer plataforma de middleware disponível na biblioteca de plataformas alvo, e.g. ROS. Esse script deve ser colocado no nó filho "MessageExchange/Software/Remote" para permitir que a ferramenta GenERTiCA gere o código fonte apropriado para o objeto.

O ROS fornece um mecanismo para facilitar a implementação da troca de informações entre objetos sejam eles locais ou remotos. O engenheiro não precisa indicar explicitamente no código se a comunicação acontece entre objetos locais ou remotos. Conforme mencionado na seção II-B, o mestre ROS é o responsável por encontrar e colocar os objetos em contato direto. Para tanto, é necessário que todos os nós ROS se registrem no mestre ROS, informando o seu identificador único, os tópicos que eles assinam/publicam e os serviços que eles fornecem. O código referente a este processo de registro é gerado juntamente com o código de inicialização do ROS. A informação de identificação do nó ROS é gerada a partir do nome da classe seguida pelo caractere "_" e pelo nome do objeto. Caso o engenheiro não tenha especificado o nome do objeto explicitamente, um numero aleatório é gerado e concatenado na string que identifica o nó ROS.

Os nós ROS se comunicam sincronamente através de chamadas de serviço ou assincronamente através do envio/recepção de mensagens em tópicos ROS. Os serviços de um nó ROS representam um estrutura cliente/servidor e são implementados através das classes *ros::ServiceServer* e *ros::ServiceClient*. Durante a inicialização, o nó ROS deve avisar ao mestre ROS quais são os serviços que ele pode fornecer. Dessa forma, este trabalho propõe que todos os métodos públicos de um objeto (com exceção os métodos *get/set*) sejam mapeados como serviços ROS. Assim, para cada método público um objeto *ros::ServiceServer* é criado e inicializado na função *main()*. A Figura 4 mostra o script responsável por gerar o código dos serviços fornecidos pelo nó ROS. O trecho em vermelho apresenta o código que registra o serviço no mestre ROS. Cada serviço é identificado através nome da classe, seguido pelo nome do objeto, e o nome método. Por outro lado, as informações dos parâmetros de entrada e retorno (conforme o especificado no modelo UML/MARTE) são usadas para gerar o código relativo ao formato de dados enviados/recebidos que aparecem em verde.

```
#foreach ($meth in $Class.getMethods())
  #if ( $meth.isPublic() && !$meth.isGetSetMethod()) )
    set( $SRV_NAME = "${OBJECT_NAME}_${meth.Name}")
    /* Service Input/Output parameters. Copy/paste the next
       lines into a file named $SRV_NAME.srv
       ***********************************************
       # request input data
       #foreach( $param in $meth.Parameters )
          #if ( $param.isInput() )
            $param.DataType $param.Name
          #end
       #end
       ---
       # response data
       #foreach( $param in $meth.Parameters )
          #if ( $param.isOutput() )
            $param.DataType $param.Name
          #end
       #end
       ***********************************************
     */
    ros::ServiceServer $SRV_NAME =
     ${OBJECT_NAME}.advertiseService("$SRV_NAME", ${meth.Name});
  #end
#end
```

Fig. 4. Script para a geração de código de serviços dos nós ROS servidores

Por sua vez, todos os objetos que chamam serviços de outros objetos devem implementar os clientes destes serviços. Seguindo a semântica do ROS, é necessário criar um objeto que representa esse cliente. Então deve-se procurar todas as ações que fazem a chamada de métodos para identificar se há ou não chamadas a serviços de outros objetos. Conforme mencionados, métodos *get/set* são ignorados pois são mapeados diretamento para tópicos ROS. A Figura 5 apresenta o script responsável por criar os clientes de serviços ROS chamados dentro de um nó ROS. O texto em verde mostra o script que faz a busca por ações de chamadas de métodos de outros objetos/classes, enquanto o texto em verde o código gerado para o objeto cliente do serviço ROS.

```
#foreach ($meth in $Class.getMethods())
  #if ( $meth.isPublic() && !$meth.isGetSetMethod())
    #foreach ($element in
                  $meth.TriggeredBehavior.BehavioralElements)
      #if ($element.isSendMessageAction
           && $element.ToObject.InstanceOf != $Class )
        set( $SRV_NAME =
                    ${element.ToObject.InstanceOf.Name}+"_"+
                    ${element.ToObject.Name}+"_"+
                    ${element.RelatedMethod.Name} )

        ros::ServiceClient "${SRV_NAME}_client" = ${OBJECT_NAME}
              .serviceClient<uml-to-ros::$SRV_NAME>("$SRV_NAME");
      #end
    #end
  #end
#end
```

Fig. 5. Script para a geração de código de chamada dos serviços dos nós ROS clientes

A comunicação assíncrona no ROS é feita através da publicação/recepção de mensagens nos tópicos ROS. Estas ações são implementadas pelas classes *ros::Publisher* e *ros::Subscriber*. O envio de mensagens é feito através da chamada do método *ros::Publisher.publish()* e a recepção de mensagens através de uma função de *call-back*. Este trabalho propõe que sejam criados tópicos ROS para cada atributo que tenha métodos *get()* ou *set()* públicos. Assim, é necessário gerar um código de registro dos tópicos para que o mestre ROS possa fazer a distribuição correta dos dados enviados nas mensagens associadas aos tópicos ROS. A Figura 6 apresenta o script que gera o código de registro da assinatura (texto em roxo) e publicação dos tópicos ROS (texto em vermelho).

```
#foreach ($meth in $Class.Methods )
  #if ( $meth.isPublic() && $meth.isSetMethod() )
    set( $TOPIC_NAME = $OBJECT_NAME+"_"+
                       $meth.AssociatedAttribute.Name )
    /* Message format. Copy/paste the next lines
       into a file named ${TOPIC_NAME}.msg
       $meth.AssociatedAttribute.DataType)
                          $meth.AssociatedAttribute.Name
       ## TODO generate of complex datatype e.g. classes
     */
    ros::Publisher ${TOPIC_NAME}_pub = ${OBJECT_NAME}.advertise
                <uml-to-ros::$TOPIC_NAME>("$TOPIC_NAME", 1000);

  #elseif ( $meth.isPublic() && $meth.isGetMethod() )
    ros::Subscriber ${TOPIC_NAME}_sub = ${OBJECT_NAME}.subscribe
                ("${TOPIC_NAME}", 1000, ${TOPIC_NAME}_call_back);
  #end
#end
```

Fig. 6. Script para a geração de código de registro de tópicos ROS assinados e publicados por um nó ROS (associados aos métodos *get/set* de atributos)

Este trabalho propõe a inclusão de uma chamada ao método *ros::Publisher.publish()* toda vez que um método *set()* for chamado, além de uma função call-back para cada método *get()*. Por sua vez, as ações de chamada dos método *get/set* dentro dos comportamentos dos objetos são ignoradas durante a geração de código, pois não são mais necessárias devido ao mecanismo de tópicos do ROS. A Figura 7 mostra o script que gera as funções de call-back referentes aos método *get()* públicos associados aos atributos dos objetos.

```
#foreach ($meth in $Class.Methods )
  #if ( $meth.isPublic() && $meth.isGetMethod() )
    void ${TOPIC_NAME}_call_back(const
                        uml-to-ros::$TOPIC_NAME::ConstPtr& msg) {
      ${OBJECT_NAME}.${meth.AssociatedAttribute.Name} =
                        msg->data;
    }
  #end
#end
```

Fig. 7. Script para a geração de código das funções de call-back referentes aos método get() públicos associados aos atributos dos objetos

Por sua vez, a Figura 8 apresenta o script responsável por gerar o código para as chamadas de métodos. O texto em preto mostra o script que gera as chamadas a métodos do próprio objeto. O texto em vermelho mostra a chamada a serviços ROS disponibilizados por outros nós ROS que podem representar objetos locais ou remotos. É possível notar que o código gerado usa o método *ros::ServiceClient.call()* para realizar a chamada ao serviço ROS. Por fim, o texto em verde mostra o script que substitui a chamada a um método *set()* pelo método *ros::Publisher.publish()* que publica uma mensagem no tópico associado ao atributo do objeto.

```
#if ($Action.getToObject() == $Action.getFromObject())
  ${Action.RelatedMethod.Name}(
  #if ($Action.ParametersValuesCount > 0)
    #foreach($param in $Action.getParametersValues())
      #if ($velocityCount > 1), #end
      #set( $x =  $velocityCount - 1)
      #if ($Action.isParameterValue($x))
        ${param}
      #else
        ${param.Name}
      #end
    #end
  #end
  );
#elseif ( $Action.RelatedMethod.isPublic()
          && !$meth.isGetSetMethod() )
  ## code for calling ROS service
  set( $SRV_NAME = ${${$Action.ToObject.InstanceOf.Name}+"_"+
                    ${$Action.ToObject.Name}+"_"+
                    ${$Action.RelatedMethod.Name} )
  <uml-to-ros::$SRV_NAME> srv;
  #foreach( $param in $Action.RelatedMethod.Parameters )
    #if ($velocityCount > 1), #end
    #set( $x =  $velocityCount - 1)
    #if ( $param.isInput() )
       srv.${param.Name} =
       #if ($Action.isParameterValue($x))
         ${param};
       #else
         ${param.Name};
       #end
    #end
  #end
  ${SRV_NAME}_client.call(srv);
#elseif ( $Action.RelatedMethod.isPublic()
          && $meth.isSetMethod() )
  ## code for publishing on a ROS topic
  set( $TOPIC_NAME = ${OBJECT_NAME}+"_"+
          ${${$Action.RelatedMethod.AssociatedAttribute.Name}" )
  uml-to-ros::$TOPIC_NAME msg;
  msg.data = $Action.RelatedMethod.AssociatedAttribute.Name;
  ${TOPIC_NAME}_pub.publish(msg);
#end
```

Fig. 8. Script para a geração de código das funções de call-back referentes aos método *get()* públicos associados aos atributos dos objetos

É importante ressaltar que tanto para a criação de serviços como também mensagens/tópicos, é necessário criar arquivos adicionais contendo o formato dos dados enviados. No caso dos serviços é necessário descrever a quantidade e o tipo dos parâmetros de entrada do serviço, bem como a quantidade e o tipo de dados que serão retornados como resposta. No caso das mensagens, é necessário informar a quantidade e o tipo de dados que serão enviados/recebidos através dos tópicos. É necessária a inclusão destes arquivos nos arquivos de configuração da compilação/construção do repositório ROS. A ferramenta *catkin_make* interpreta estes arquivos e gera os arquivos de cabeçalho C/C++ contendo as estruturas de dados correspondentes. No entanto, a versão atual da ferramenta GenERTiCA não tem a capacidade de gerar estes arquivos de forma separada. Então, quando ela gera o código de definição dos serviços e tópicos ROS, a definição do formato de dados é gerada na forma de comentários que podem ser facilmente copiados em arquivos separados (ver texto em verde nas Figuras 4 e 6). Um trabalho futuro é estender a ferramenta GenERTiCA de modo que ela possa gerar automaticamente esses arquivos auxiliares. Além disso é importante suportar a descrição de formatos complexo de mensagens que incluem múltiplos dados com tipos diferentes – não apenas os tipos primitivos suportados até o momento.

A tabela I apresenta um resumo dos mapeamentos criados entre UML/MARTE e ROS. Destaca-se que este trabalho não fornece o mapeamento para o servidor de parâmetros do ROS.

No entanto, os conceitos principais (e.g. nós, tópicos, mensagens, e serviços) são plenamente suportados.

TABLE I. MAPEAMENTO UML-PARA-ROS

UML/MARTE	ROS
Classes	Definição das classes, função *main()*
Objetos	Nós ROS (processos), código de registro no mestre ROS
<<SchedulableResource>>	Estrutura de repetição *while*
<<TimedEvent>>	ros::Rate
Atributos	Tópicos e Mensagens ROS
Métodos *get()*	funções *call-back*, ros::Subscriber, ros::NodeHandle.*subscribe()*
Métodos *set()*	ros::Publisher, ros::NodeHandle.*advertise()*
Chamada de métodos *set()*	ros::Publisher.*publish()*
Métodos públicos	ros::ServiceServer
Chamada de métodos públicos	ros::ServiceClient, ros::ServiceClient.*call()*

IV. ESTUDO DE CASO: ROBÔ SEGUIDOR DE LINHA

A abordagem de geração de código proposta neste trabalho foi validada através do projeto de um robô seguidor de linha [3] implementado com o ROS. O robô seguidor de linha é composto por: (i) dois sensores infravermelho – um para o lado esquerdo e outro para o lado direito; (ii) duas rodas; (iii) dois servo-motores. O robô se desloca para frente ao longo de um trajeto marcado no chão com uma linha preta. Os sensores infravermelhos estão posicionados de tal forma que a linha preta fica posicionada entre eles sem ser detectada. Caso o robô se desloque para cima dessa linha, um dos sensores de infravermelho detecta tal ação. Então, o controlador corrige a rota através da atuação nos motores que controlam as rodas direita e esquerda, fazendo com que o robô realize curvas.

O sistema de controle do robô foi inicialmente projetado em [3] e adaptado no contexto deste trabalho. A Figura 9 mostra o diagrama de classes criado. A classe *MovementControl* é responsável pelo controle do movimento do robô, fazendo-o mover-se para a frente seguindo o trajeto marcado no chão. A classe *Motor* é responsável por gerar o sinal PWM para os servo-motores que, por sua vez, controlam o giro das rodas direita e esquerda. Por fim, a classe *VelocityControl* é responsável por controlar a velocidade dos motores e implementar uma rampa de velocidade.

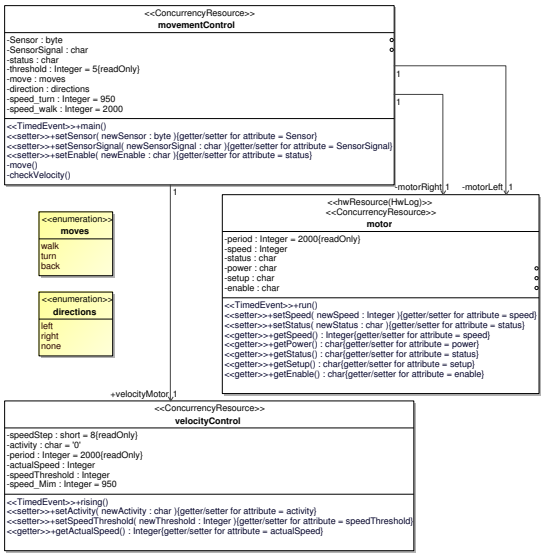

Fig. 9. Diagrama de classes do robô seguidor de linha

Foram implementados quatro nós ROS. A Figura 10 apresenta o código gerado para o nó ROS que implementa o objeto *movctrl* da classe *MovementControl*. Este código foi gerado pelos script apresentados na seção anterior com base nas informações contidas no diagrama de classes mostrado na Figura 9. O texto em preto representa a definição da classe *MovementControl*. As construções e bibliotecas do ROS necessárias para implementar o robô seguidor de linha são mostradas em texto vermelho e verde. O texto em verde representa os tópicos associados a atributos dos objetos *motorLeft*, *motorRight*, e *velocityMotor*. Os métodos *main()*, *move()* e *checkVelocity()* da classe *MovementControl* publicam mensagens nestes tópicos. Os comentários gerados para indicar o formato das mensagens foram omitidos.

```
#ifndef MovementControl_h
#defin MovementControl_h

class MovementControl
{
protected:
  int Sensor
  char SensorSignal;
  char status;
  int threshold;
  ENUM_MOVES move;
  ENUM_DIRECTIONS direction;
  int speed_turn;
  int speed_walk;

  void move();
  void checkVelocity();
public
  void main();
  setSensor( int newSensor );
  setSensorSignal( char newSensorSignal );
  setEnable( char newEnable );
};

int main(int argc, char **argv) {
  ros::init(argc, argv, "MovementControl_movctrl");
  ros::NodeHandle MovementControl_movctrl;

  /* Message format. Copy/paste the next lines
     into a file named MovementControl_movctrl_Sensor.msg
     int32 Sensor
  */
  ros::Publisher MovementControl_movctrl_Sensor_pub =
     MovementControl_movctrl.advertise<uml-to-ros::
                    MovementControl_movctrl_Sensor>(
                    "MovementControl_movctrl_Sensor", 1000);
  ... // comentarios omitidos
  ros::Publisher MovementControl_movctrl_SensorSignal_pub =
     MovementControl_movctrl.advertise<uml-to-ros::
                  MovementControl_movctrl_SensorSignal>(
                  "MovementControl_movctrl_SensorSignal", 1000);
  ... // comentarios omitidos
  ros::Publisher MovementControl_movctrl_Enable_pub =
     MovementControl_movctrl.advertise<uml-to-ros::
                    MovementControl_movctrl_Enable>(
                    "MovementControl_movctrl_Enable", 1000);
  ... // comentarios omitidos
  ros::Publisher Motor_motorLeft_Status_pub =
       MovementControl_movctrl.advertise<uml-to-ros::
                      Motor_motorLeft_Status>(
                      "Motor_motorLeft_Status", 1000);
  ... // comentarios omitidos
  ros::Publisher Motor_motorRight_Status_pub =
       MovementControl_movctrl.advertise<uml-to-ros::
                      Motor_motorRight_Status>(
                      "Motor_motorRight_Status", 1000);
  ... // comentarios omitidos
  ros::Publisher Velocitycontrol_velocityMotor_Activity_pub =
       MovementControl_movctrl.advertise<uml-to-ros::
                Velocitycontrol_velocityMotor_Activity>(
                "Velocitycontrol_velocityMotor_Activity", 1000);

  MovementControl_movctrl.main();

  return 0;
}
#endif
```

Fig. 10. Código gerado para o nó ROS que representa o objeto *movctrl* da classe *MovementControl*

A Figura 11 apresenta a parte inicial do diagrama de sequencias que especifica o comportamento do método *main()* da classe *MovementControl*. Pode-se observar que o objeto *movctrl* envia o valor do seu *status* para os objetos *motorLeft*, *motorRight*, e *velocityMotor*. Após isso o algoritmo controla a direção do movimento, faz o robô mover-se e verifica se a

velocidade dos motores está adequada ao valor atual de velocidade fornecido pelo objeto *velocityMotor*. O código fonte gerado para o método *MovementControl.main()* é mostrado na Figura 12. O texto em vermelho indica o código responsável por publicar mensagens nos tópicos ROS referentes ao atributo *Status* dos objetos *motorLeft*, *motorRight*, e o atributo *Activity* do objeto *velocityMotor*. O texto em verde representa a implementação do aspecto *PeriodicTiming* [1][2] usando as construções disponíveis no ROS para executar o comportamento periodicamente de acordo com a frequência especificada através do estereótipo *<<TimedEvent>>* mostrado na Figura 11.

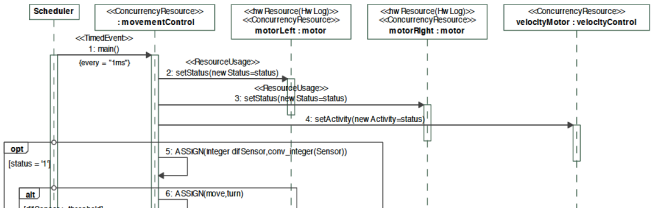

Fig. 11. Parte inicial do diagrama de sequencias do método *main()* da classe *MovementControl*

```
void MovementControl::main() {

  ros::Rate loop_rate(1000); // 1ms period = 1000 Hz
  while (ros::ok()) {

    Motor_motorLeft_Status_pub.publish(status);
    Motor_motorRight_Status_pub.publish(status);
    Velocitycontrol_velocityMotor_Activity_pub.publish(status);

    if ( status = '1' ) {
      int difSensor = Sensor;
      if ( difSensor > threshold ) {
        if ( SensorSignal = '0' ) {
          direction = LEFT;
        }
        else {
          direction = RIGHT;
        }
      }
      else {
        move = WALK;
        direction = NONE;
      }
    }
    move();
    checkVelocity();

    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

Fig. 12. Código gerado para a função *MovementControl::main()* que implementa o coportamento principail do robô seguir de linha.

A ferramenta GenERTiCA gerou um total de 6 arquivos de código C++, sendo 3 arquivos de cabeçalho e 3 arquivos de código fonte. O total de linhas de código geradas foi de 419 linhas, sendo que, em média, foram geradas 139,6 linhas por classe especificada no modelo UML/MARTE.

V. Trabalhos Relacionados

A literatura recente apresenta várias abordagens que visam a geração de código fonte para o software de sistemas robóticos. Baumgartl et al. [15] propõe uma abordagem integrada que aplica várias linguagens específicas de domínio (do inglês, *Domain Specific Languages* – DSL) no projeto de robôs pessoais. Para isso, foi implementada uma ferramenta usando o *Eclipse Modeling Framework* (EMF) que permite a geração de código C++/ROS, porém define abstrações diferentes das disponíveis no ROS. Um trabalho similar é apresentado em [17], cujo foco é a integração do conhecimento de diferentes *stakeholder* no projeto de robôs de serviço. Para tal, um meta-modelo é proposto para integrar a informação de diferentes DSLs. Um sistema executável baseado em uma arquitetura de referência é gerado a partir dessa informação. Em [19], os autores propõe uma abordagem MDE que visa encontrar as melhores soluções de projeto possíveis em tempo de execução (e, assim, alcançar tanto robustez por design como a robustez por adaptação) ao invés de encontrar as soluções ótimas em tempo de projeto. Esse trabalho propõe o uso da semântica de componentes. Um ferramental baseado no EMF e UML foi criado juntamente com um perfil UML. Complementarmente, um meta-modelo para representar os conceitos do ROS foi proposto em [20]. Segundo os autores, aquele trabalho é um passo inicial para a criação de uma abordagem MDE que suporta transformações entre modelos intermediários e também a geração de código. O meta-modelo proposto parece interessante e tem potencial para alavancar os trabalhos relacionados com MDE e ROS. Contudo, a especificação de sistema robótico não é abordada naquela trabalho.

Uma outra abordagem baseada em MDE foi proposta em [16] para o projeto de linhas de produtos de software para sistemas de percepção de robôs. A abordagem sugere o uso de uma DSL chamada *Robot Perception Specification Language* que, por sua vez, não é uma linguagem de especificação padrão, o que acaba dificultando a adoção da abordagem proposta. Além disso, não há menção sobre a possibilidade de geração de código. O trabalho proposto em [18] propõe a geração de código fonte C++ que usa ROS a partir das informações especificadas em AutomaML, que é uma linguagem similar ao XML. A abordagem proposta permite a geração de uma grande quantidade de código, além dos arquivos auxiliares utilizar pelas ferramentas do ROS para a construção do artefato executável. Apesar disso, pode-se entender como uma desvantagem o uso de uma linguagem de marcação como forma de especificar os requisitos funcionais e não-funcionais de um sistemas robótico. Adicionalmente, uma abordagem para gerar código ROS a partir de modelos AADL foi discutida em [21]. Aquele trabalho apresenta um estudo de caso detalhado de como modelar um robô complexo e fazer a geração do artefato executável.

A principal diferença dos trabalhos mencionados com o apresentado aqui é o uso de uma linguagem padrão de fato no contexto de engenharia de software, como a UML/MARTE, e o seu mapeamento direto para construções e bibliotecas do ROS. Além disso, aplicar a abordagem AMoDE-RT no projeto de sistemas robóticos permite a separação das preocupações com os requisitos funcionais e não-funcionais. Tal característica não é suportada explicitamente em nenhum dos trabalhos citados.

VI. Conclusões e Trabalhos Futuros

Este trabalho apresentou a ampliação da abordagem AMoDE-RT, que se baseia em técnicas da engenharia guiada por modelos, para o projeto do software que executa nos sistemas embarcados controlam robôs. Foi proposta uma abordagem para a geração automática do código fonte a partir das informações contidas em um modelo UML/MARTE. A implementação deste software utiliza o *Robot Operating System* (ROS) e se baseia na semântica de comunicação *publisher-subscriber* e chamada de serviços.

Para tanto, criou-se um conjunto de regras de mapeamento de elementos suportados na UML para construções e bibliotecas disponíveis no ROS. Este mapeamento foi implementado na forma de scripts de geração de código que são usados na ferramenta GenERTiCA. Cada objeto/classe

especificado no modelo UML é mapeado em um nó ROS. Os métodos públicos especificados nas classes são mapeados em serviços ROS que tem a semântica de execução síncrona semelhante a chamada remota de procedimentos. Os métodos públicos do tipo *get/set* de atributos são mapeados em tópicos ROS e permitem a comunicação assíncrona entre os nós ROS. Os nós ROS assinam e publicam mensagens nestes tópicos. Quando um objeto chama um método *set(),* o código fonte correspondente a essa ação publica uma mensagem no tópico ROS relacionado ao atributo no objeto alvo. Por sua vez, os métodos *get()* são mapeados em funções *call-back* que são executados toda vez que uma nova mensagem é recebida pelo nó ROS e faz o recebimento da mensagem e a devida atribuição do novo valor no atributo em questão.

A abordagem proposta foi validada através de um estudo de caso: o projeto de um robô seguidor de linha. Apesar de simples, este projeto possibilitou avaliar a viabilidade da abordagem proposta para a geração de código do software de robôs implementados usando o ROS. Foi possível gerar em média 139,6 linhas de código C++ por classe e um total de 419 linhas para as classes que implementam o robô seguidor de linhas. Tais resultados representam um indicativo importante sobre a viabilidade da aplicação da abordagem proposta em sistemas mais complexos. Acredita-se fortemente que é possível criar modelos mais complexos de sistemas robóticos e gerar grande parte do código necessário para a sua implementação usando o ROS.

Como trabalhos futuros, pretende-se realizar a geração de código de projetos mais complexos de sistemas robóticos que envolvem vários robôs e.g. uma célula de manufatura inteligente ou um robô autônomo aéreo. Além disso, é necessário modificar e estender a ferramenta GenERTiCA de modo que ela possa gerar automaticamente arquivos auxiliares como os arquivos de definição do formato de mensagens dos tópicos ROS e arquivos de configuração das ferramentas de construção do ROS. Além disso é importante criar regras de mapeamento para suportar a descrição de formatos complexo de mensagens que incluem múltiplos dados com tipos diferentes e.g. objetos com todos os seus atributos. Outra direção de trabalhos futuros é fornecer o suporte as preocupações transversais presentes em uma aplicação ROS através da implementação de regras de mapeamento para outros aspectos disponíveis no DERAF.

AGRADECIMENTOS

O autor agradece o apoio financeiro da Fundação Araucária de Apoio ao Desenvolvimento Científico e Tecnológico do Estado do Paraná através dos projetos 337/2014 e 34/2019 e da Universidade Tecnológica Federal do Paraná (UTFPR) através do Edital UTFPR/IPB 01/2017.

REFERENCES

[1] M. A. Wehrmeister, C. E. Pereira and F. J. Rammig, "Aspect-Oriented Model-Driven Engineering for Embedded Systems Applied to Automation Systems," in IEEE Transactions on Industrial Informatics, vol.9, no.4, pp. 2373-2386, Nov. 2013. doi: 10.1109/TII.2013.2240308

[2] M.A. Wehrmeister, et al. “Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems,” in Mechatronics, Vol. 24, no. 7, pp. 844-865, Oct. 2014. doi: 10.1016/j.mechatronics.2013.12.008

[3] M. Leite, M. A. Wehrmeister, “System-level design based on UML/MARTE for FPGA-based embedded real-time systems,” in Design Automation for Embedded Systems, Vol. 20, no. 2, pp. 127-153, Jun. 2016. doi: 10.1007/s10617-016-9172-6

[4] Object Management Group (OMG), Unified Modeling Language version 2.5.1, 2017, https://www.omg.org/spec/UML/2.5.1/

[5] Object Management Group (OMG), UML Profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE), version 1.2, 2018, https://www.omg.org/spec/MARTE/

[6] Robert Filman, et al. Aspect-Oriented Software Development. 1st ed. Addison-Wesley Professional, 800 p. 2004.

[7] E.P. de Freitas, et al., “DERAF: A High-Level Aspects Framework for Distributed Embedded Real-Time Systems Design”, in Early Aspects: Current Challenges and Future Directions. Lecture Notes in Computer Science, vol 4765, pp 55-74, 2007. doi: 10.1007/978-3-540-76811-1_4

[8] M.A. Wehrmeister, E.P. de Freitas, C.E. Pereira. “An Infrastructure for UML-Based Code Generation Tools” In: Analysis, Architectures and Modelling of Embedded Systems. IFIP AICT, vol.310. pp.32-43, 2009, doi: 10.1007/978-3-642-04284-3_4

[9] M. A. Wehrmeister, E. P. Freitas, C. E. Pereira and F. Rammig, "GenERTiCA: A Tool for Code Generation and Aspects Weaving," 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), Orlando, FL, 2008, pp. 234-238. doi: 10.1109/ISORC.2008.67

[10] R. Rajkumar, M. Gagliardi and Lui Sha, "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation," Proc. Real-Time Technology and Applications Symposium, Chicago, IL, USA, 1995, pp. 66-75. doi: 10.1109/RTTAS.1995.516203

[11] A.P.D. Binotto, et al. “Sm@rtConfig: a Context-Aware Runtime and Tuning System using an Aspect-Oriented Approach for Data Intensive Engineering Applications”, in Control Engineering Practice, vol. 21, no. 2 p. 204-217, Feb. 2013. doi: 10.1016/j.conengprac.2012.10.001

[12] T. G. Moreira, et al. "Automatic code generation for embedded systems: From UML specifications to VHDL code," 2010 8th IEEE International Conference on Industrial Informatics, Osaka, 2010, pp. 1085-1090. doi: 10.1109/INDIN.2010.5549590

[13] Open Source Robotics Foundation., Robot Operating System (ROS), http://www.ros.org

[14] L. Joseph, and J. Cacace, Mastering ROS for Robotics Programming - Second Edition: Design, build, and simulate complex robots using the Robot Operating System. 2018. Packt Publishing Ltd.

[15] J. Baumgartl et al. "Towards easy robot programming: Using DSLs, code generators and software product lines”, Proc. of the 8th Int. Joint Conference on Software Technologies, Reykjavik, 2013, p. 548-554.

[16] D. Brugali And N. Hochgeschwender. “Software Product Line Engineering for Robotic Perception Systems”. International Journal of Semantic Computing, v. 12, n. 01, p. 89-107, 2018.

[17] Kai Adam, et al. “Model-driven separation of concerns for service robotics”. In: Proc. of the International Workshop on Domain-Specific Modeling. ACM, 2016. p. 22-27

[18] Y. Hua, et al. "From AutomationML to ROS: A model-driven approach for software engineering of industrial robotics using ontological reasoning," 2016 IEEE 21st Int. Conf. on Emerging Technologies and Factory Automation (ETFA), Berlin, 2016, pp. 1-8.

[19] C. Schlegel, et al. Design abstraction and processes in robotics: From code-driven to model-driven engineering. In: Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots. Springer, Berlin, Heidelberg, 2010. pp. 324-335.

[20] N. Hammoudeh Garcia, et al. , "Bootstrapping MDE Development from ROS Manual Code - Part 1: Metamodeling," 2019 IEEE Int. Conf. on Robotic Computing (IRC), Naples, Italy, 2019, pp. 329-336.

[21] G. Bardaro, et al. “A use case in model-based robot development using AADL and ROS”. In: Proceedings of the International Workshop on Robotics Software Engineering. ACM, 2018. p. 9-16.

[22] E. Estévez, et al. “A novel model-driven approach to support development cycle of robotic systems”. International Journal of Advanced Manufacturing Technology, v. 82, n. 1-4, p. 737-751, 2016.

[23] M. Wenger, et al "A model based engineering tool for ROS component compositioning, configuration and generation of deployment information," 2016 IEEE 21st Int. Conf. on Emerging Technologies and Factory Automation (ETFA), Berlin, 2016, pp. 1-8.

[24] N. GOBILLOT, Nicolas, et al.. “A Design and Analysis Methodology for Component-Based Real-Time Architectures of Autonomous Systems”. Journal of Intelligent & Robotic Systems, p. 1-16, 2018.