BY-NC-SA
https://doi.org/10.17979/spudc.9788497498081.0467

GENERACIÓN AUTOMÁTICA DE CÓDIGO DE APLICACIONES ROBÓTICAS MANIPULADORAS PARA EL MIDDLEWARE DE ROS

E. Estévez, A. Sánchez García, J. Gámez García, J. Gómez Ortega
Escuela Politécnica Superior de Jaén.
Universidad de Jaén
Email: {eestevez, asgarcia, jggarcia, juango}@ujaen.es

Resumen

Actualmente, los robots manipuladores están presentes en la mayor parte de instalaciones de producción industrial moderna. Asimismo, dicho tipo de robots también están siendo incorporados en muchos más entornos e.g. realización de tareas domésticas, asistencia domiciliaria... Es por ello que a día de hoy, existe una demanda creciente de aplicaciones con robots manipuladores más flexibles, adaptables a diferentes entornos y focalizando a su vez en la realización de código. Sin embargo, hay una carencia de estandarización de plataformas hardware y software, por lo que es extremadamente complicado satisfacer dichas demandas. Este trabajo explora las ventajas proporcionadas por la Ingeniería Dirigida por Modelos (MDE) en el diseño y desarrollo de tareas realizadas por robots manipuladores. Concretamente se propone una herramienta que permite modelar una tarea robótica manipuladora abstrayendo al experto de dominio de tener que conocer la plataforma donde se vaya a ejecutar dicha aplicación y generar automáticamente el código final para ROS (Robotic Operating System), uno de los middleware de comunicación más utilizados en esta disciplina.

Palabras Clave: Robots manipuladores, MDE, ROS-Robotic Operating System

1 INTRODUCCIÓN

Actualmente, la robótica manipuladora es una disciplina decisiva en las instalaciones de producción industriales y muy pronto, también lo será en la vida cotidiana de la sociedad. Por todo ello, existe una demanda creciente de aplicaciones cada vez más complejas y con requisitos como la *reutilización*, *flexibilidad* y *adaptabilidad*. Lamentablemente, pese a la gran variedad de robots en el mercado, el desarrollo de software en campos específicos como la robótica, está más cerca de ser considerado arte que una disciplina sistemática [9]. Esto fundamentalmente se debe a :

1) la gran variabilidad de aplicaciones así como componentes hardware/software (HW/SW);
2) elevada dificultad de conseguir una reutilización real debido al gran número de elementos (dispositivos manipuladores, algoritmos de procesamiento, middleware de comunicaciones …);
3) falta de interoperabilidad entre las herramientas involucradas en las diferentes fases del ciclo de desarrollo de las aplicaciones.

Para poder asegurar el cumplimiento de dichos requisitos, las arquitecturas HW/SW deberán permitir a los desarrolladores hacer frente a la complejidad impuesta por las propias aplicaciones, hardware, software, requisitos temporales y entornos de computación distribuidos.

Para reducir esta complejidad, se ha ido evolucionando en el uso de tecnologías de la ingenería del software [17] En este contexto, este trabajo explora las ventajas que proporciona el uso de ingeniería dirigida por modelos (MDE) [7], [3], para dar soporte al ciclo de desarrollo de aplicaciones robóticas manipuladoras. En los últimos años, se ha comenzado a introducir esta disciplina en el campo de la robótica [8]. Por ejemplo, en [5] se presenta una herramienta llamada EasyLab basada en dos lenguajes gráficos propietarios de modelado: *Synchronous Data Model* muy similar a Sequential Function Chart de IEC 61131-3 para definir la funcionalidad del sistema. El segundo lenguaje de modelado está centrado en la descripción del hardware, como una colección de sensores y actuadores, pero sin ninguna representación ni nomenclatura estándar. Dicha herramienta, inicialmente, estaba enfocada a sistemas mecatrónicos; aunque posteriormente los autores la adaptaron para plataformas Robotino Mobile Robot© [11]. En [2] se presenta un lenguaje de modelado que contempla la descripción de aplicaciones robóticas desde tres puntos de vista: (1) *estructural* para definir la estructura estática de la aplicación basada en componentes; (2) *coordinación* donde se define el comportamiento de cada componente y (3) *algoritmo* para la definición del código ejecutado en cada componente en función de su estado. Dichas vistas están definidas a través de un conjunto de diagramas UML (Unified Modeling Language) y se generan código Ada para CORBA. Más recientemente, se ha desarrolado la herramienta MDSD Toolchain [20]

dentro del marco del proyecto SmartSoft, inspirada en el estándar Model Driven Architecture [13]. El proyecto BRICS (Best Practices in Robotics) [6] también proporciona una herramienta llamada BRIDE para facilitar el diseño de las aplicaciones robóticas [22]. En dicha herramienta hay que decantarse inicialmente por el MW (Middleware) sobre el que se ejecutará el código e.g. OROCOS o ROS (Robotic Operating System) y en función de la selección permite interconectar gráficamente componentes OROCOS o Nodos ROS. En este sentido, permite una reutilización de código entre aplicaciones que se ejecuten sobre una misma plataforma.

Pese a que todos los trabajos comentados previamente hacen uso del diseño basado en modelos, el presente trabajo pretende ir un paso más allá ya que presenta una herramienta basada en MDE donde por un lado permite al experto de dominio modelar la funcionalidad de toda aplicación robótica manipuladora y además fomenta la reutilización de código de una manera independiente a la plataforma de ejecución. De los trabajos previamente citados, BRIDE tiene un objetivo similar pero la reutilización de código se asegura entre aplicaciones que se ejecutan en la misma plataforma (OROCOS o ROS). Para conseguir una reutilización de código independiente a la plataforma, hace falta de una herramienta que por un lado permita modelar la funcionalidad de una aplicación y, posteriormente, especificar la plataforma sobre la que se va a ejecutar. Dicha herramienta ha de finalizar con el soporte a la generación automática de código. Por ello, la herramienta BRIDE no es válida para este objetivo.

Para el modelado de la funcionalidad de toda aplicación robótica manipuladora, los autores en [18] identificaron y caracterizaron las interfaces de los diferentes componentes que se pueden dar (sensores, actuadores y algoritmos de control). Para especificar la lógica de la aplicación así como la plataforma, los autores han definido una herramienta basada en Graphiti [12] al que se le ha añadido la generación automática de código a ROS. Para ello se han hecho uso de técnicas Model to Text (M2T) [14] de MDE.

La estructura del trabajo es la siguiente: en la sección 2 se describe las pautas propuestas para el modelado de aplicaciones robóticas manipuladoras. La sección 3 en primer lugar describe el editor gráfico y posteriormente las reglas de transformación de código para generar aplicaciones que se ejecuten sobre ROS. La sección 4 presenta un caso de estudio de robótica manipuladora industrial. Finalmente, la sección 5 presenta las conclusiones del artículo.

2 MODELADO DE LAS APLICACIONES ROBÓTICAS MANIPULADORAS

Trabajos previos como [4] y [10] han identificado y caracterizado los tipos de componentes que participan en las tareas de robots manipuladores siendo estos: los *sensores*, *actuadores* y *algoritmos de procesamiento* como generadores de trayectorias.

La Figura 1 ilustra a través de un diagrama de clases UML las características comunes a los tipos de componentes identificados. Todas ellas, se encuentran agrupadas en AtomicTask, que proporciona como propiedad común la frecuencia de ejecución [18]. Así, por ejemplo, en el caso de los sensores esta propiedad es fundamental para indicar cada cuánto se requiere una medida. En el caso de los robots y algoritmos de control, esta propiedad se utiliza para indicar cada cuánto tiempo se han de ejecutar. Para los sensores se añaden como características la naturaleza de la magnitud a medir (*type*), número de muestras (*size*) así como el valor de la medida (*value*). En función de la magnitud a medir los sensores requerirán de propiedades y métodos específicos. Por ejemplo, en el caso de capturar imágenes se añadirán las propiedades de imagen adquirida, anchura y altura. Además hay que resaltar que cada sensor de cámara específica (véase parte inferior de la Figura 1, Guppy80, GX1050 y SR 4000) añadirá particularidades del fabricante a través de una serie de propiedades y métodos privados.

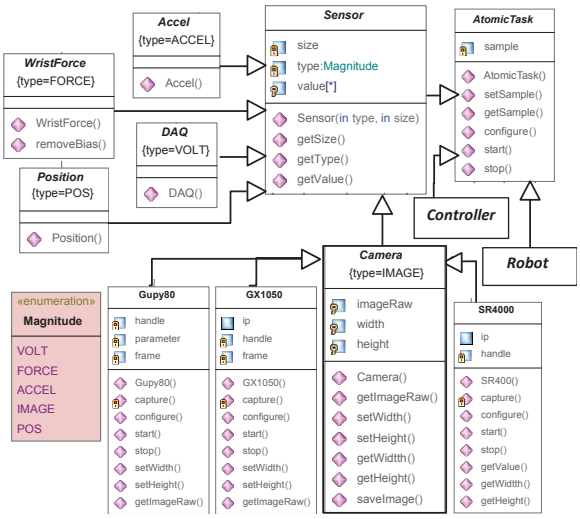

Figura 1: Caracterización de los módulos de las tareas robóticas manipuladoras

Los autores en trabajos previos [18] han identificado y caracterizado los tipos de componentes que participan en las tareas de robots manipuladores (sensores, manipuladores y algoritmos de procesamiento como generadores de trayectorias). Las interfaces propuestas por un lado ayudan a los codificadores a añadir nuevos componentes a la base de datos (donde se almacenan las unidades mínimas de codificación) y por otro lado proporcionan una

abstracción total a los diseñadores del manejo de los drivers propios de cada fabricante.

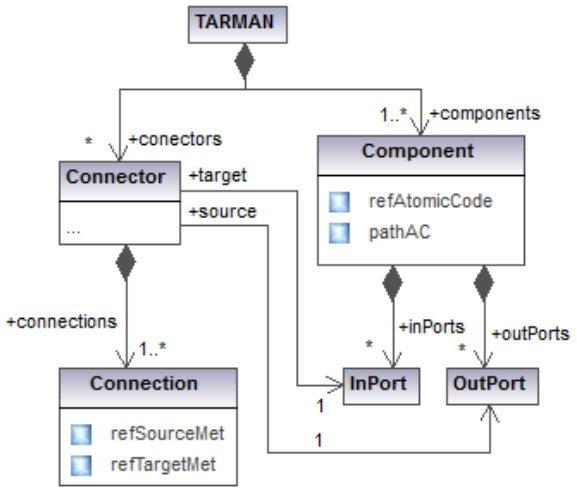

Figura 2: Meta-modelo de funcionalidad de Tarea Robótica MANipuladora

En la fase de diseño se modela la funcionalidad/lógica de las aplicaciones así como la plataforma (HW/SW) donde se desplegará el código generado.

La Figura 2 presenta el meta-modelo resumido, que recoge la funcionalidad de cualquier tarea robótica manipuladora. Toda Tarea Robótica MANipuladora (TARMAN) queda definida por un conjunto de componentes interconectados. Cada componente encapsula un código atómico (independiente a la aplicación) y a través de los puertos define la accesibilidad al mismo (con permisos de lectura-*OutPort*-, o escritura-*InPort*-). Así, a través de los puertos de entrada/salida queda definida la interfaz de los componentes. La lógica de la aplicación queda definida a través de los conectores que interconectan componentes a través de sus puertos. Los conectores son una colección de conexiones, siendo éstas últimas las que indican la información intercambiada. El instante de tiempo en el que se actualiza la información a intercambiar viene determinado por la propiedad *sample* del código atómico encapsulado (véase Figura 1). Si dicha propiedad tiene un valor diferente a cero es que se trata de un módulo que genera y consulta información de forma periódica. De lo contrario si su valor es cero, se trata de un módulo que responde bajo demanda (evento).

3 TARMANtool: HERRAMIENTA DE MODELADO Y GENERACIÓN AUTOMÁTICA DE CÓDIGO

En este apartado se presenta la herramienta *TARMANtool* desarrollada haciendo uso de técnicas de la ingeniería dirigida por modelos, que permite definir la funcionalidad de toda tarea robótica manipuladora y empleando técnicas M2T genera el código final de la misma para el Middleware ROS.

3.1 Editor Gráfico

A la hora de construir un editor gráfico para un modelo de dominio (Véase Figura 2) a través de Eclipse, Eclipse Modeling Framework (EMF) y Graphical Editing Framework (GEF) son dos plugins muy utilizados.

Por un lado, EMF es una herramienta que proporciona la base para el modelado y facilidades para la generación de código con objeto de construir herramientas u otras aplicaciones basadas en un modelo de datos estructurado. A partir de una especificación XML de un modelo, EMF suministra herramientas y soporte de ejecución para producir un conjunto de clases Java basadas en ese modelo, un conjunto de clases Adapter, que permiten su visualización y edición basándose en comandos del modelo, y un editor básico. EMF permite importar modelos que han sido previamente especificados usando documentos Ecore, este tipo de documentos pueden definirse como el metamodelo usado por EMF para construir la herramienta. Una de las características importantes de Ecore es que los metamodelos y modelos se representan en archivos XML, y además son capaces de generar código automático a partir del metamodelo definido. La siguiente figura ilustra el Ecore definido acorde a la Figura 2 para la herramienta TAMANtool.

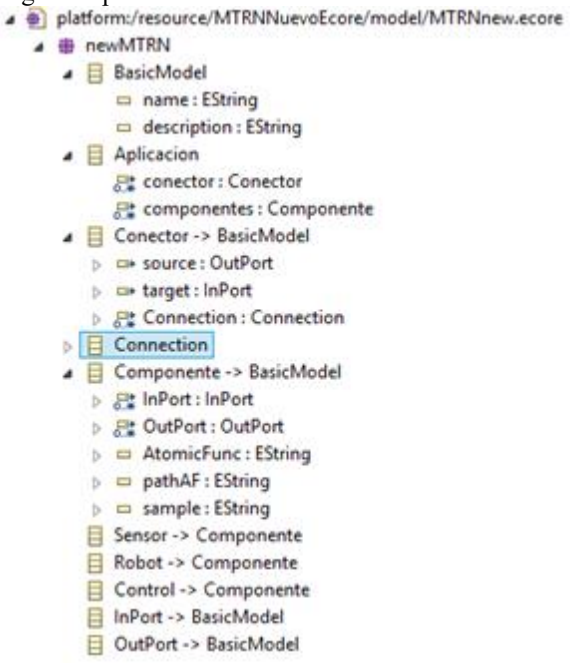

Figura 3: Meta-modelo Ecore de la Herramienta

Dicho meta-modelo está compuesto por:

- *BasicModel*: entidad abstracta de la que heredan los demás elementos. Los atributos básicos de de todos los elementos son:

- ❖ name: es de tipo String y se utiliza para que cada entidad tenga su propio nombre.
- ❖ description: se utiliza para añadirle a la entidad una breve descripción que ayude a comprender cuál es su función. Es de tipo String.

- *Aplicacion*: entidad raíz de todas las entidades de la herramienta. Esta entidad representa la aplicación en sí, ya que contiene las conexiones y los componentes. Está compuesta de dos tipos referencias:
  - ❖ conector: referencia a Conector, con una multiplicidad de 0 a muchos, que da a entender, que la aplicación puede contener ninguno o varios Conectores.
  - ❖ componentes: referencia a Componente, misma multiplicidad que la anterior. La aplicación podrá contener 0 o varios componentes.
- Conector: esta entidad representa el conector de la aplicación, hereda de la entidad BasicModel, y será la encargada de crear las conexiones entre los diferentes componentes de la aplicación. Consta de tres tipos referencias:
  - ❖ source: referencia a OutPort, y refleja el puerto origen del conector.
  - ❖ Target: referencia a InPort, refleja el puerto destino del conector en la aplicación.
  - ❖ Connection: referencia a la entidad Connection, tiene una multiplicidad de 0 a muchos. Significa que cada conector puede contener a su vez de 0 a varias conexiones.
- Connection: representa la conexión, puede ser descrita como la lógica interna del conector y dispone de dos atributos:
  - ❖ SourceFunction: este atributo es de tipo String y se utiliza para dar nombre a la función de origen que va se va a conectar.
  - ❖ TargetFunction: se utiliza para dar nombre a la función de destino que va a ser conectada con la función origen.
- Componente: entidad abstracta que es la raíz de los componentes de la aplicación, engloba los atributos y referencias básicos de cada uno de los componentes de la aplicación. A su vez esta entidad hereda de BasicModel los atributos *name* y *description*. La entidad se compone de los siguientes elementos:
  - ❖ InPort: referencia a entidad InPort, con multiplicidad de 0 a muchos, a diferencia de la referencia que se produce en la entidad Conector, ésta define la cantidad de puertos de entrada que cada componente puede contener. Y como se observa en la multiplicidad la cantidad podrá ser de cero a varios.
  - ❖ OutPort: referencia a OutPort, multiplicidad de 0 a muchos, esta referencia hace lo mismo que la anterior, pero en este caso con los puertos de salida.
  - ❖ AtomicFunc: atributo de tipo String que vincula una funcionalidad atómica a un componente de la aplicación.
  - ❖ pathAF: este atributo es el encargado de almacenar el path de la funcionalidad atómica vinculada a cada componente, para permitir acceder a dicha funcionalidad de una manera directa. Es de tipo String.
  - ❖ sample: atributo de tipo String que identifica diferentes tipos de comportamiento dependiendo del componente que lo utiliza.

Por otro lado GEF proporciona la tecnología para crear editores gráficos. Dada la complejidad del manejo de GEF, en el 2009 la multinacional SAP AG donó a la comunidad de Eclipse el pluging de Graphiti [12]. Dado que Graphiti apoya la creación rápida y fácil de los editores gráficos homogéneos este trabajo se apoya en dicho pluging. Un middleware de desarrollo ágil es Spray [21]. Los usuarios podrán visualizar un modelo de dominio subyacente utilizando una notación gráfica de la herramienta definida.

La comunicación entre un usuario y la herramienta Graphiti se lleva a cabo a través de la interfaz. Los dos componente de Graphiti encargados de dar las funcionalidades a los usuarios son: (1) *Interaction Component* que recibirá peticiones para cambiar el tamaño, arrastrar y soltar, o borrar del editor y (2) *Rendering Engine* responsable de mostrar la información y realizar el procesamiento de las peticiones que realiza el usuario a través del *Diagram Type Agent*.

La principal tarea de *Diagram Type Agent* es modificar los datos cuando el usuario comienza a interactuar con el editor. Para esto hace uso de los siguientes tres modelos:

- Modelo de dominio (Domain Model): contiene los elementos del meta-modelo que tienen que ser visualizados gráficamente. Es el que se diseña con EMF a través del código generado en base al metamodelo Ecore. Define la lógica de la aplicación.
- Modelo Pictograma (Pictogram Model): contiene la información completa para representar el editor gráfico. Los diagramas pueden ser representados sin la presencia de los datos del domino. Esto obliga a que se almacenen los datos en este modelo y de manera redundante en el modelo de domino.
- Modelo de Enlace (Link Model): es el responsable de enlazar los datos del modelo de domino con la representación gráfica o modelo pictograma.

Con la ayuda de las herramientas creadas con Spray, un editor Graphiti se puede crear de una manera

mucho más rápida y fiable a través de los siguientes ficheros:

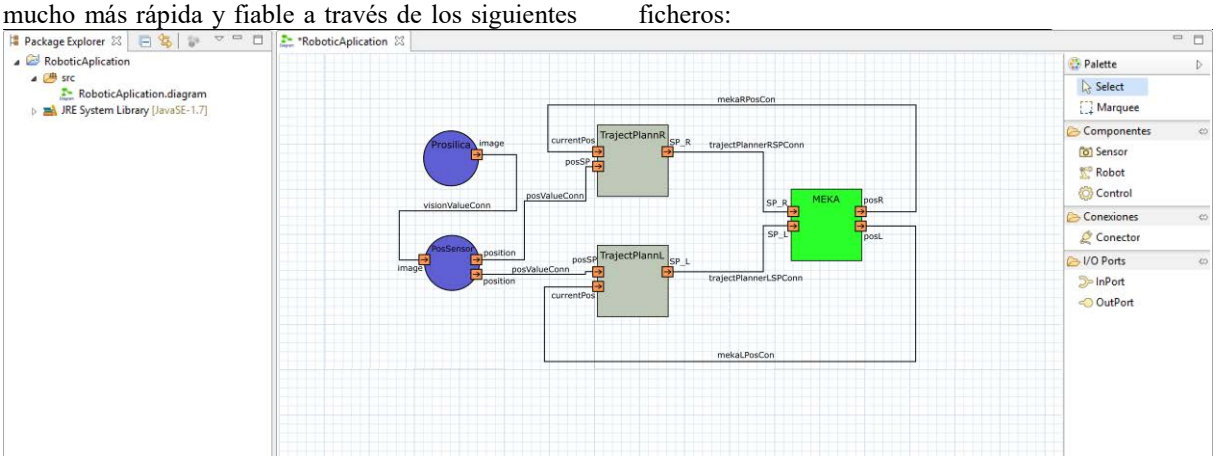

Figura 4: Ejemplo de modelado de una tarea robótica manipuladora

- '.style'. Define el comportamiento del color y formato de los componentes del diagrama. Se comporta de manera similar a una hoja de estilos en HTML, utilizando un lenguaje muy similar.
- ´.shape'. Se utiliza para definir las formas y las conexiones. Estas formas se generan a partir de otras formas primitivas ya definidas, como rectángulos y elipses. También se configuran otros aspectos como el tamaño y la superposición para generar figuras más complejas.
- '.spray'. Es el responsable de enlazar el metamodelo de la aplicación con las formas, estilos y comportamientos del editor gráfico.

Finalmente, la Figura 4 ilustra un ejemplo de modelado gráfico de una tarea robótica manipuladora.

3.2 Generador de código para ROS

La plataforma ROS pese a que no está basada en componentes, está formada por unidades modulares de programación llamadas nodos. La comunicación entre los nodos se puede llevar a cabo con los modelos de comunicación: *publicista/suscriptor* o *cliente/servidor*.

Para que sea publicista/suscriptor están involucrados los tópicos y los mensajes. Un nodo que quiera hacer accesible un dato, publica un tópico. De igual manera cuando un nodo requiera de una información se ha de subscribir a dicho tópico.

Por otro lado, cuando la comunicación es cliente/servidor, el nodo que actúa de servidor se queda a la espera de recibir una petición por parte del nodo cliente. Cuando el nodo cliente realiza la solicitud, el nodo servidor realiza un procesamiento (servicio) y responde al cliente. Por lo que en este caso se intercambian dos mensajes: uno de petición y otro de respuesta. En realidad dicha interacción se presenta como la llamada a un procedimiento remoto.

A continuación se listan las principales reglas de transformación identificadas a seguir para la generación de código de los nodos ROS. En itálica aparece resaltada la información necesaria del fichero generado en el apartado anterior (TARMAN.xml):

- Regla 1: Generación de los nodos ROS de aplicación. Se genera uno por cada componente TARMAN.
- Regla 2: Datos a publicar. Se realiza una búsqueda por cada puerto de salida del componente TARMAN. Aquel conector cuya Fuente tenga como identificador el "*componente[@id] /Outport[@id]*" facilita toda la información para publicar un dato. Así, por cada dato a publicar será necesario definir
  - Un dato protegido de tipo publicador.
    Si es de tipo básico [16]:
    ros::Publisher msg*Conector[@id]*;
    Si es de tipo imagen (ROS sensor_msgs, 2015):
    image_transport::ImageTransport Transp*Conector[@id]*;
    image_transport::Publisher iPub*Conector[@id]*;
  - Método responsable de publicar el tópico:
    void Publish*Conector[@id]*();

  Cada tópico a publicar ha de ser inicializado en el constructor indicando el tipo de dato. Posteriormente, el método responsable de realizar la publicación del tópico, en primer lugar define una variable auxiliar del mismo tipo que el dato (message) a publicar, posteriormente se le asigna el valor del dato, y finalmente se publica.
- Regla 3: Datos a suscribirse. Se realiza una búsqueda por cada puerto de entrada del componente TARMAN. Aquel conector cuyo Destino tenga como identificador el "*componente[@id]/Inport[@id]*" facilita toda la información para suscribirse a un dato. Por cada dato a suscribirse será necesario definir:
  - Un dato protegido de tipo suscriptor.
    Si es de tipo básico:

ros::Subscriber msg*Conector[@id]*;

Si es de tipo imagen:

image_transport::ImageTransport Transp*Conector[@id]*;

image_transport::Publisher iPub*Conector[@id]*;

cv_bridge::CvImagePtr iCvConector[@id];

Método responsable de suscribirse al tópico:

void *Conector[@id]*CallBack (const TipoDato::ConstPtr& message);

Cada tópico a suscribirse ha de ser inicializado en el constructor.

Para ello a la función susbcribe hay que pasarle como parámetros: el nombre del tópico a suscribirse, número de datos que se almacenan en el buffer (1) y el método que accede a la información del tópico al que se ha suscrito el nodo (*Conector[@id]*CallBack). Esta función se encarga de actualizar el dato con el valor del tópico al que se ha suscrito el nodo.

- Regla 4: Generación de la función principal del nodo ROS. Se ha definido una estructura fija común para todos los nodos.
  - En primer lugar se ha de inicializar el nodo: ros::init (argc, argv, RosNode*Componente[@id]*();
  - Posteriormente es necesaria la definición de un manejador de nodo para poder comunicarse con el ROS Master [1].
  - Para terminar con las definiciones, se define un objeto de la clase que representa el nodo de aplicación ROS a generar: rosNode*Componente[@id]* rosNode(n); En esta definición se invoca al constructor por defecto, al que se le facilita el manejador de nodo (*n*).
  - Si el nodo ROS va a publicar datos, hace falta indicar cada cuánto va a refrescar la información publicada: ros::Rate loop_rate(*Componente[@id]/@sample*);
  - La lógica del nodo se implemente por medio de un bucle donde:
    1) se ejecuta la lógica que encapsula: rosNode.*Update*();
    2) se publican todos los tópicos: rosNode.publish*Conector[@id]*();
    3) se procesan los mensajes por parte del ROS Master: ros::SpinOnce();
    4) se duerme al nodo hasta que pase el tiempo para volver a mandar: loop_rate.sleep();

Una vez generados todos los códigos fuente de los nodos que participarán en la aplicación robótica manipuladora, el siguiente paso es compilarlos. Para ello se ha de definir un *ros package* con el nombre de la *aplicación* dentro de la zona de trabajo (ROS_workspace/src). Una vez creado dicho paquete, todos los ficheros .h y .cpp se almacenarán en *aplicación/src*.

Finalmente, en el fichero *CMakeLists.txt* por cada nodo a compilar se ha de añadir: Rosbuild_add_executable(RosNode*Componente[@id]* src/Ros RosNode*Componente[@id]*.cpp src/Ros RosNode*Componente[@id]/@funcionalidad*.cpp).

Una vez compilada la aplicación, para arrancarla se puede hacer a través de un fichero launch [1] donde se indican los nodos ROS a arrancar para poner en marcha la aplicación generada. La siguiente figura ilustra un ejemplo. Por cada nodo a arrancar, hay que indicar el paquete donde se encuentra el tipo de nodo que en este tipo de aplicaciones coincide con el nombre del mismo.

launch

node (5)

	= pkg	= type	= name
1	TrackingObject	RosNodeCamera	RosNodeCamera
2	TrackingObject	RosNodePosSensor	RosNodePosSensor
3	TrackingObject	RosNodeTrajecPlannR	RosNodeTrajecPlannR
4	TrackingObject	RosNodeTrajecPlannL	RosNodeTrajecPlannL
5	TrackingObject	RosNodeMeka	RosNodeMeka

Figura 5: Ejemplo de fichero launch

4 CASO DE ESTUDIO: SEGUIMIENTO DE UN OBJETO EN MOVIMIENTO.

El seguimiento de objetos es una tarea cotidiana que los humanos realizan fácilmente, pero cuando es realizada por un robot no es una tarea trivial por varios motivos. Por una parte, es necesario tener un reconocimiento completo del entorno donde se mueve el robot pudiendo haber en ocasiones problemas de oclusión de objetos. Por otro lado, una vez localizado el objeto, se ha de definir un control de trayectorias para poder seguirlo de manera apropiada, evitando cualquier tipo de colisión con otros objetos incluidos en la escena [15].

Este caso de estudio describe cómo el robot humanoide Meka realiza la tarea de seguimiento con los dos brazos de un objeto en movimiento, lo cual, implica no solamente la localización 3D del objeto sino también el control de trayectoria de los dos brazos para evitar cualquier tipo de colisión.

Para esta tarea de seguimiento, se ha utilizado como sensor extereoceptivo un sensor de visión (Prosilica GX1050) que tiene como objetivo, junto con algoritmos de procesamiento de imágenes, determinar la posición 3D del objeto. Para simplificar el caso de estudio, se partirá de un entorno parcialmente conocido en el que tanto la ubicación de los obstáculos como la cinta transportadora es conocida. La tarea comienza, por tanto, reconociendo el objeto a seguir. Posteriormente, el robot Meka va ajustando la posición de los brazos en función de la posición 3D facilitada por el sistema de visión por computador.

Figura 6: seguimiento de un objeto en movimiento por parte del robot Meka

La Figura 6 ilustra una secuencia de movimientos del robot Meka. Como se observa, en este caso para la localización 3D del objeto se ha hecho uso de una cámara convencional, se ha dotado al objeto de un patrón conocido que junto con el algoritmo de procesado de 4 puntos permite la localización del objeto. La Figura 4 ilustra la funcionalidad de la aplicación que correrá sobre el MW de ROS.

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include "Prosilica.h"
class RosNodeCamera  : public  Prosilica]{
  protected:
      ros::Handle node;
      image_transport::ImageTransport  iTranspVisionValueConn;
      image_transport::Publisher iPubVisionValueConn;
public:
                  RosNodeCamera ();
                  virtual RosNodeCamera();
      void PublishVisionValueConn();
};
```

RosNodeCamera.h

```
#include "RosNodeCamera.h"
RosNodeCamera::RosNodeCamera(ros::NodeHandle n_): node(n_){
                  //publicar una imagen
                  iPubVisionValueConn =
iTranspVisionValueConn.advertise("VisionValueConn",1);
}
RosNodeCamera::PublishVisionValueConn(){
    cv_bridge::CvImagePtr message;
    message->image= getImageRaw();
                  iPubVisionValueConn.publish(message->toImageMsg());
}
int main (int argc, char** argv){
                  ros::init(argc,argv, RosNodeCamera);
                  ros::NodeHandle n;
                  RosNodeCamera rosNode(n);
                  ros::Rate loop_rate(20.0);
                  while(ros::ok()){
                                    rosNode.Update();
                                    rosNode.PublishVisionValueConn(); // por cada tópico
a publicar
                                    ros::spinOnce();
                                    loop_rate.sleep();
                  }
                  return 0;
}
```

RosNodeCamera.cpp

Figura 7: Código fuente del nodo ROS camera

La Figura 7 detalla el código generado automáticamente para la definición del nodo ROS cámara. Como se puede apreciar, publica 20 imágenes por segundo. Para lanzar la aplicación generada también se genera el fichero launch con la lista de nodos a arrancar (véase Figura 5).

5 CONCLUSIONES.

Este trabajo presenta una herramienta gráfica para dar soporte al ciclo de desarrollo a aplicaciones robóticas manipuladoras que se ejecuten sobre ROS, haciendo uso de los principios de MDE. En concreto, se han fijado unas pautas de diseño, en las que se asegura una reutilización de código entre aplicaciones que se ejecutan sobre una mismo middleware e incluso entre aplicaciones que también se ejecutan en MW diferentes.

Cabe destacar que el hecho de generar el código final en dos pasos, permite desacoplar el diseño de la generación en sí. Como resultado del diseño de una aplicación se dispone internamente del modelo TARMAN.xml que es el punto de partida para la generación del código final (M2T). Para añadir la generación de un nuevo MW únicamente sería necesario desarrollar el transformador M2T correspondiente. Los autores también generan código para aplicaciones robóticas manipuladoras que se ejecuten sobre el MW de OROCOS.

Agradecimientos

Los autores quieren agradecer la subvención parcial de este trabajo a través de los proyectos DPI2011-27284, TEP2009-5363 y AGR-6429.

Referencias

[1] Aaron Martinez, Enrique Fernández (2013), "Learning ROS for Robotics Programming", Packt Publishing ltd.

[2] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor y B. Álvarez (2010), "V3CMM: a 3-view component metamodel for model-driven robotic software development", Journal of Software Engineering for Robotics, pp: 3-17.

[3] K. Balasubramanian, A. Gokhale, G. Karsai, J.Sztipanovits, y S. Neema, (2006) "Developing applications using model-driven design environments,"Computer, vol. 39, no. 2, pp. 33 – 40.

[4] Bárbara Álvarez, Francisco Ortiz, Juan A Pastor, Pedro Sánchez, Fernando Losilla, Noelia Ortega, (2006), "Arquitectura para control de robots de servicio teleoperados". Revista Iberoamericana de Automática e Informática Industrial. Vol:3(2), pp:79-89.

[5] S. Barner, M. Geisinger, C. Buckl, y A. Knoll, (2008) "EasyLab: model-based development

ofsoftware for mechatronic systems," in IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications, pp. 540 –545.

[6] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, J. Broenink, D. Brugali, y N. Tomatis, (2010), "BRICS – best practice in robotics," 41st International Symposium on and 6th German Conference on Robotics (ROBOTIK), pp. 1 –8.

[7] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, y A. Oreback (2005), "Towards component-based robotics," IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 163 – 168.

[8] D. Brugali y A. Shakhimardanov (2010), "Component-based robotic engineering (Part II) [Tutorial]," Robotics Automation Magazine, IEEE, vol. 17, no. 1, pp.100 –112

[9] Chella, A., Cossentino, M., Gaglio, S., Sabatucci, L., Seidita, V., (2010). "Agent oriented software patterns for rapid and affordable robot programming". Journal of Systems and Software 83, Issue:4, pp:557 – 573.

[10] Gabriel J. Garcia, Juan A. Corrales, Jorge Palomares, Fernado Torres (2009), "Survey of Visual and Force/Tactile Control of Robots for Physical Interaction in Spain", Sensors, Vol:9, pp:9689-9733.

[11] Michael Geisinger, Simon Barner, Martin Wojtczyk, y Alois Knoll (2009), "A software architecture for model-based programming of robot systems," LNCS, Advances in Robotics Reseachr, Springer, pp. 135–146

[12] Graphiti (2016). [online] https://eclipse.org/graphiti/documentation/

[13] J. Miller y J. Mukerji, "Mda guide versión 1.0.1," 2003. [Online]. Disponible en: http://www.enterprisearchitecture-info/Images/MDA/

[14] OMG. Meta Object Facility (MOF) 2.x XMI Mapping Specification. [Online] Disponible en: http://www.omg.org/spec/XMI/

[15] Poza-Lujan, J. L., Posadas-Yagüe, J. L., Simó-Ten, J. E., Simarro, R., & Benet, G. (2015). Distributed sensor architecture for intelligent control that supports quality of control and quality of service. Sensors, 15(3), 4700-4733.

[16] ROS msg, 2015, [Online] http://wiki.ros.org/msg

[17] I. Sommerville (2010), "Software Engineering", 9th ed. Addison-Wesley.

[18] .A. Sanchez-Garcia, E. Estevez, J. Gomez Ortega, J. Gamez Garcia. (2013) "Component-based modelling for generating robotic arm applications running under OROCOS middleware" IEEE International Conference on Systems, Man, and Cybernetics. pp 3633-3638.

[19] Alejandro Sánchez García, Jesús De La Casa Cárdenas, Elisabet Estévez, Javier Gámez García, Juan Gómez Ortega y Silvia Satorres, (2015), "Uso de MDE para el diseño y codificación automática de plataformas robóticas manipuladoras", XXXVI Jornadas de Automática, pp: 221-229.

[20] C. Schlegel, A. Steck, D. Brugali, y A. Knoll, (2010) "Design abstraction and processes in robotics: From code-driven to model-driven engineering," in Simulation, Modeling, and Programming for Autonomous Robots, LNCS, Eds. Springer Berlin/Heidelberg, vol. 6472, pp. 324–335.

[21] Spray (2016). https://code.google.com/a/eclipselabs.org/p/spray/

[22] D. Steinberg, F. Budinsky, M. Paternostro y E. Merks, (2008), EMF: Eclipse Modeling Framework, 2nd ed. Addison-Wesley Professional.

[23] Tidwell (2001), D. XSLT, Ed. O'REILLY.