ScienceDirect
Disponible en www.sciencedirect.com
Revista Iberoamericana de Automática e Informática industrial 14 (2017) 82–93
www.elsevier.es/RIAI

Aproximación Basada en UML para el Diseño y Codificación Automática de Plataformas Robóticas Manipuladoras

Elisabet Estévez [*], Alejandro Sánchez García, Javier Gámez García, Juan Gómez Ortega

Grupo de Robótica, Automática y Visión por Computador (GRAV), Universidad de Jaén, Paraje Las Lagunillas s.n., 23071, Jaén, España

Resumen

Hoy en día, los robots manipuladores están presentes en todas las instalaciones de producción moderna industriales. Por ello, a la robótica manipuladora se la considera una disciplina decisiva en el sector industrial. Además, en un futuro no muy lejano los robots manipuladores pasarán a ser también esenciales en la vida cotidiana de la sociedad. Así, existe una demanda creciente de aplicaciones con robots manipuladores con requisitos software como son la reutilización, flexibilidad y adaptabilidad. Lamentablemente, en la actualidad hay una falta de estandarización de plataformas hardware y software, por lo que es extremadamente complicado satisfacer estos requisitos. Así, se contempla, por lo tanto, una necesidad de definir una metodología que proporcione unas pautas y guíe en el diseño, implementación y en la ejecución de sistemas software para este tipo de aplicaciones. Este trabajo explora las ventajas proporcionadas por la Ingeniería Dirigida por Modelos (MDE- Model Driven Engineering) en el diseño y desarrollo de tareas realizadas por robots manipuladores. En concreto se propone una aproximación basada en un sub-conjunto de diagramas UML (diagrama de componentes y diagrama de despliegue) para el diseño de este tipo de aplicaciones. Además, se identifican y definen los pasos a realizar para la generación automática de código que se ejecutará sobre los middlewares (MW) de comunicación más utilizados en esta disciplina. Para ello, se hará uso de las técnicas Model-To-Model y Model-To-Text nuevamente de MDE. Este trabajo, detalla la generación del código para OROCOS (el MW basado en componentes más extendido) y ROS. Finalmente, se presentan dos casos de estudio, el primero de ellos consiste en una aplicación industrial para el ensamblado de faro de vehículos que se ejecuta sobre OROCOS. El segundo caso de estudio es una aplicación de robótica de servicios en la que un robot manipulador antropomórfico realiza la tarea de seguimiento de un objeto en movimiento. Este segundo caso de estudio se ejecuta sobre ROS.

Palabras Clave:
Robots manipuladores, UML, MDE-Model Driven Engineering, ROS-Robotic Operating System-, OROCOS – Open Robot Control Software-.

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 (Aracil et al., 2008). Por todo ello, existe una demanda creciente de aplicaciones cada vez más complejas y con requisitos como la reutilización, flexibilidad y adaptabilidad (Valera et al., 2012).

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 (Chella et al., 2010). 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…) y (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.

En este contexto, este trabajo explora las ventajas que proporciona el uso de ingeniería dirigida por modelos (Model Driven Engineering - MDE) (Balasubramanian et al., 2006), (Brooks et al., 2005) para dar soporte al ciclo de desarrollo a aplicaciones robóticas manipuladoras. En los últimos años, se ha comenzado a introducir esta disciplina en el campo de la robótica (Brugali and Shakhimardanov, 2010). Por ejemplo, en (Barner et al., 2008) 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

* Autor en correspondencia.
Correo electrónico: eestevez@ujaen.es (Elisabet Estévez)

© 2016 CEA. Publicado por Elsevier España, S.L.U. Este es un artículo Open Access bajo la licencia CC BY-NC-ND (http://creativecommons.org/licenses/by-nc-nd/4.0/)
http://dx.doi.org/10.1016/j.riai.2016.11.001

ninguna representación ni nomenclatura estándar. Dicha herramienta, inicialmente, estaba enfocada a sistemas mecatrónicos aunque posteriormente los autores la adaptaron para plataformas Robotino Mobile Robot© (Michael et al., 2009). En (Alonso et al., 2010) 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-(Booch eta al., 2005) y se generan código Ada para CORBA. Más recientemente, se ha desarrollado la herramienta MDSD Toolchain (Schlegel et al., 2010) dentro del marco del proyecto SmartSoft, inspirada en el estándar Model Driven Architecture (Miller and Mukerji, 2003). El proyecto BRICS (Best Practices in Robotics) (Bischoff et al., 2010) también proporciona una herramienta llamada BRIDE (Garcia and Bruyninckx, 2014) para facilitar el diseño de las aplicaciones robóticas que está basado en EMF- Eclipse Modeling Framework (Steinberg et al., 2008). En dicha herramienta hay que decantarse inicialmente por el MW sobre el que se ejecutará el código (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: tiene como objetivo especificar unas pautas de diseño de aplicaciones robóticas manipuladores, fomentando la reutilización de código de una manera independiente a la plataforma de ejecución y además haciendo uso de una notación estándar. 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 (Sanchez-Garcia et al., 2013) 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 seleccionado UML como notación estándar. Concretamente, a través del diagrama de componentes UML se modela la funcionalidad de la aplicación. Será a través de un diagrama de despliegue UML donde se especifique la plataforma en la que se ejecutará el código generado. Además, se identifican y definen los pasos a realizar para la generación automática, nuevamente haciendo uso de técnicas de MDE (Atkinson et al., 2003), (Selic, 2003).

Una primera versión de modelado se realizó en (Estévez et al., 2015). El presente trabajo añade la definición de un perfil UML con objeto de proporcionar independencia de herramienta de modelado UML en la generación de código. De esta manera toda herramienta UML que soporte importar/exportar ficheros XMI (XML Metadata Interchange, 2014) se podrá utilizar como herramienta de modelado (e.g. Papyrus, Altova Umodel, Rational Rose…). Además realiza la generación de código en dos pasos: Model To Model (M2M) y Model to text (M2T). Con M2M se filtra la información en formato XMI exportada de la herramienta de modelado dejando la aplicación como un conjunto de componentes interconectados y con M2T se realiza la generación de código en sí para los MW de comunicaciones basados en componentes así como para ROS.

La estructura del trabajo es la siguiente: en la sección 2 se detallan los componentes de las aplicaciones robóticas. La sección 3 describe las pautas propuestas para modelar en UML este tipo de aplicaciones. La sección 4 se centra en la generación automática de código particularizándola para aquellos MW basados en componentes y ROS. La sección 5 presenta dos casos de estudio uno de robótica manipuladora industrial y otro de servicios. Finalmente, la sección 6 muestra las conclusiones y trabajos futuros.

2. Modelado de los componentes de las aplicaciones robóticas manipuladoras

Trabajos previos como (Bárbara et al., 2006) y (Gabriel et al., 2009) 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 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. 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 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. (Sanchez-Garcia et al., 2013) detalla dicha caracterización.

Por otro lado, para completar el interface de cada uno de dichos módulos, se ha seguido el paradigma de Orientación a Objetos, por lo que se dispone un método get y/o set para poder consultar y/o actualizar el valor de las propiedades protegidas. Además, todo módulo constará de los métodos configure, start y stop que definen su estado cuando se despliegue en una plataforma determinada. Finalmente, hay que resaltar, que las interfaces propuestas ayudan por un lado a los programadores a añadir nuevos componentes a la base de datos y por otro lado a los diseñadores de las aplicaciones robóticas manipuladoras, ya que proporcionan una abstracción total al manejo de los drivers propios de cada fabricante.

La Figura 2a presenta un ejemplo para el caso de captura de imágenes, donde la información del fabricante está resaltada en morado y la información manejable por los diseñadores en verde. Como se puede apreciar independientemente al tipo de cámara los diseñadores manejan la misma información.

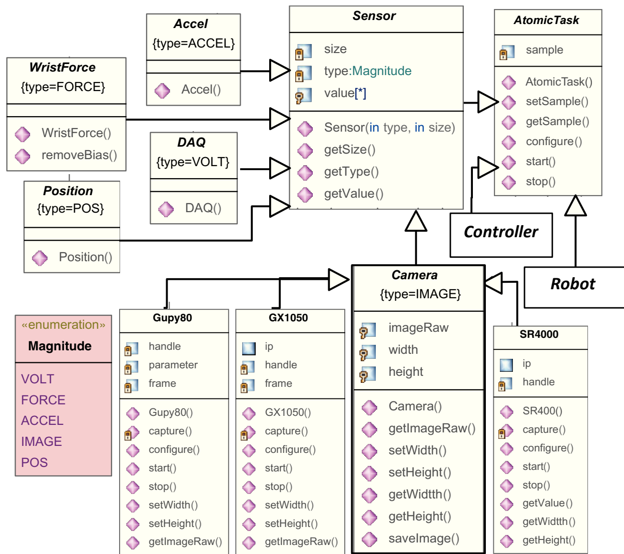

Figura 1: Caracterización de los módulos de las tareas robóticas manipuladoras

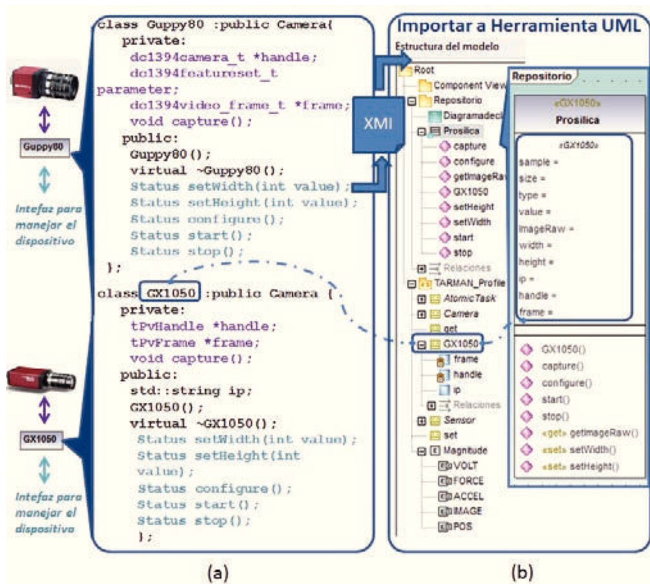

Figura 2: (a) Interface de Guppy F-080C y Prosilica GX1050 (b) representación en UML

Con objeto de especificar dichas características en UML se ha definido un perfil con tantos estereotipos como conceptos presentados en la Figura 1. Cada estereotipo estará a su vez compuesto por los valores etiquetados presentados como propiedades en las clases de la Figura 1. Además, se ha añadido el estereotipo de *get* y *set* para proporcionar accesibilidad a los parámetros (véase Figura 2b).

3. Modelado de aplicaciones robóticas manipuladoras

Este apartado propone una plataforma de modelado para dar soporte al ciclo de desarrollo de las aplicaciones robóticas manipuladoras. La Figura 3 ilustra el escenario general de dicha plataforma. Una vez realizada la especificación de requisitos funcionales y no funcionales, durante la fase de análisis se seleccionan las estrategias de control y también se almacena las unidades atómicas de código en una base de datos. 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.

Los siguientes sub-apartados detallan los pasos a seguir para dar soporte a la fase de diseño. Destacar que durante la fase de análisis (paso 1 de la Figura 3) se testea el código atómico y se almacena en el repositorio. Se considera código atómico al software mínimo que implementa una funcionalidad determinada y que es independiente a la aplicación e.g. código para obtener una imagen (*Gupy80*, *GX1050* y *SR4000* de la Figura 1). Como se ha comentado anteriormente, una completa identificación y caracterización de los componentes atómicos se detalla en el trabajo previo de los autores (Sanchez-Garcia et al., 2013).

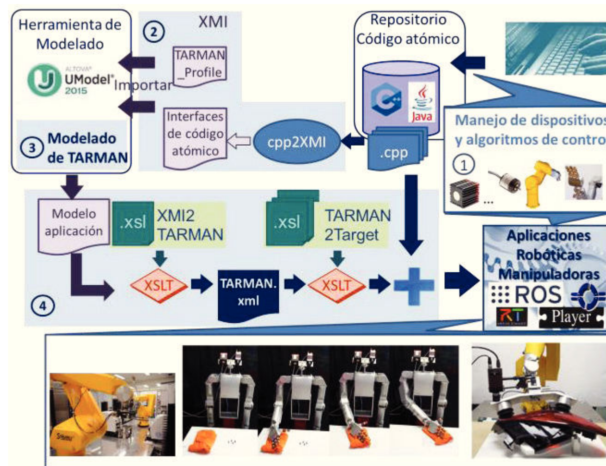

Figura 3: Escenario general de la aproximación

3.1. *Definir el funcionamiento de la aplicación*

Este paso es necesario para proporcionar a los diseñadores la mínima información requerida para poder definir una aplicación a realizar por un robot manipulador (paso 2 de la Figura 3). Por ello, se centra en importar las interfaces del código atómico de aquellos componentes básicos que participan en una tarea de este tipo (sensores, algoritmos de procesamiento…). La notación estándar elegida para representar esta información es XMI (XML Metadata Interchange Specification, 2014) ya que es la vía estándar para poder importar información a cualquier herramienta de modelado UML. Este estándar tiene una notación de marcado que permite representar los conceptos de Orientación a Objetos (OO) en XML.

Por lo tanto, para poder importar las interfaces del código atómico a una herramienta UML, han de estas expresadas en XMI. Para ello, se ha desarrollado un módulo software (*cpp2XMI*, véase Figura 3) que implementa las transformaciones ilustradas en la Tabla 1.

La Figura 2b ilustra, ya importado en Altova UModel, la interfaz para la cámara prosilica GX1050 junto con el perfil UML TARMAN (*Tareas Robóticas MANipuladoras*). La interfaz del código almacenado en el repositorio para manejar una cámara prosilica GX1050 está formado por los métodos que aparecen remarcados en verde en la Figura 2a. Nótese que a la clase UML se ha añadido el estereotipo <<*GX1050*>> lo cual implica tener

una serie de propiedades ya prefijadas e.g. *sample*, *size*, *ImageRaw*… Además, a cada método vinculado al manejo de propiedades protegidas se le ha añadido el estereotipo <<*get*>> o <<*set*>> en función de la accesibilidad que se le vaya a proporcionar a los parámetros.

Tabla 1: representación en XMI de conceptos básicos de OO

OO	notación XMI
Clase	<packagedElement xmi:type="uml:Class" xmi:id="id" name="NombreClase" isAbstract="true"/>
Propiedad	<ownedAttribute xmi:type="*uml:Property*" xmi:id="id" name="NombrePropiedad" visibility="public| protected| private"/>
Método	<ownedOperation xmi:type="*uml:Operation*" xmi:id="id" name="NombreMetodo" visibility="public | protected | private"/>
Parámetro	<ownedParameter xmi:type="*uml:Parameter*" xmi:id="id" name="NombreParam"/>
Herencia	<generalization xmi:type="*uml:Generalization*" xmi:id="id" general="id_Class1"/>

3.2. Definir el funcionamiento de la aplicación

Para definir el funcionamiento de las aplicaciones, se ha seleccionado el diagrama de componentes UML donde cada componente UML encapsula código atómico añadiéndole la lógica de la aplicación. La Tabla 2 resume los conceptos UML necesarios para modelar la lógica de las aplicaciones.

Tabla 2: Elementos UML para modelar la lógica de una aplicación

concepto UML	Notación gráfica	Rol en el modelado
Componente	«component» Comp_Name	Componente de la aplicación.
Clase + estereotipo	«GX1050» Prosilica (desde Root::Repositor	Código encapsulado en el componente (*clase base*)
Realización Componente	«component» Cámara	Forma de indicar qué código encapsula el componente
Puerto	Port1 «component» Comp_Name Port2	Proporciona accesibilidad externa a las propiedades protegidas
Interfaz		
Proporcionada (*InterfaceRealization*)	nt» Name Port2 Interface1	Acceso externo de lectura (método *get* de la clase base).
Requerida (*Usage*)	Interface2 Port1 «component» Comp_Nam	Acceso externo de escritura (método set de la clase base)

3.3. Definir el funcionamiento de la aplicación

Este apartado está enfocado al modelado de la plataforma HW y SW donde posteriormente se desplegará y correrá el código generado.

Para ello, se hace uso del diagrama de despliegue UML. La Tabla 3 resume los conceptos UML necesarios para modelar la plataforma donde se desplegará y correrá el código generado.

Tabla 3: Elementos UML para modelar la arquitectura de la plataforma

Concepto UML	Notación gráfica	Role en el modelado
Dispositivo	«device» Device1	Dispositivo (e.g. Cámara).
Nodo	Node1	Nodo (e.g. PC).
Path de comunicación	«device» Device1 Protocol Node1	Protocolo de comunicación entre los nodos y los dispositivos .
Artefacto	Node1 «artifact» Library	Librería con código atómico
Entorno de ejecución	«executionEnvironment» Comm. Middleware	Especifica el MW de comunicación.

4. Generación automática de código

Esta sección está centrada en dar soporte a la fase de generación de código (paso 4 de la Figura 3) siguiendo las directrices de las transformaciones de MDE. Concretamente, la generación de código se realiza a través de las transformaciones M2M y M2T. La primera de ellas tiene como modelo de entrada el fichero XMI exportado de la herramienta de modelado UML, que será procesado y filtrado para generar el fichero TARMAN.xml.

El modelo TARMAN.xml será la entrada a la transformación M2T para la generación automática de código. En esta segunda transformación es donde se tiene en cuenta la plataforma en la que se ejecutará el código a generar.

Los siguientes sub-apartados detallan el procedimiento seguido en dichas transformaciones.

4.1. M2M – Transformación XMI a TARMAN

En este apartado describe el procesamiento la información exportada de la herramienta de modelado UML expresada según el estándar XMI. Dicho procesamiento, consiste presentar una aplicación como un conjunto de componentes interconectados.

Las tablas Tabla 4 y Tabla 5 ilustran cómo se representan en XMI los conceptos utilizados a lo largo de la fase de diseño para modelar la lógica de la aplicación y la plataforma HW/SW, respectivamente.

Así, tal y como ilustra la Tabla 4, toda aplicación será un conjunto de componentes UML, interconectados a través de sus interfaces UML. El código encapsulado en cada componente está indicado en atributo *realizingClassifier* del elemento *realization.* Para saber si un puerto es de entrada o salida se ha de consultar el concepto *Usage*.

En lo referente a la plataforma, se especifica el hardware a través de un conjunto de dispositivos y nodos (véase Tabla 5). La comunicación entre ellos, como se ha comentado anteriormente, se especifica por medio del *path* de comunicación (*communicationPath*). Por medio de un artefacto UML se indica la librería donde se ha de almacenar el código generado. Finalmente, a través del concepto de entorno de ejecución (*ExecutionEnvironment*) queda especificado el MW de comunicaciones.

Tabla 4: Notación XMI de la funcionalidad de las aplicaciones

Modulo Funcional	
packagedElement	
xmi:type	uml:Component
xmi:id	id
name	Comp_Name
ownedAttribute	xmi:type=uml:Port xmi:id=id name=Port_Name visibility=protected
realization	xmi:type=uml:ComponentRealization xmi:id=id realizingClassifier=Class_Id
Información que proporciona el módulo funcional al exterior	
packagedElement	
xmi:type	uml:Interface
xmi:id	id
name	interface_Name
ownedOperation	xmi:type=uml:Operation xmi:id=id name=op_Name visibility=public
Información que requiere el módulo funcional del exterior	
packagedElement	
xmi:type	uml:Usage
xmi:id	id
supplier	xmi:idref=interface_id_ref
client	xmi:idref=port_id_ref

Tabla 5: Notación XMI de la plataforma Hardware y Software

Nodo	
packagedElement	
xmi:type	uml:Node
xmi:id	Node_id
name	Node_name
deployment	
xmi:type	uml:Deployment
xmi:id	id
deployedArtifact	Artifact_id
client	xmi:idref=Node_id
supplier	xmi:idref=Artifact_id
Dispositivo	
packagedElement	
xmi:type	uml:Device
xmi:id	Device_id
name	Device_name
Protocolo de comunicaciones	
packagedElement	
xmi:t...	uml:Association
xmi:id	Association_id
name	Protocol_name
ownedEnd (2)	

	xmi:type	xmi:id	visib...	type	association
1	uml:Property	Property1_id	protected	Node_id	association xmi:idref=Association_id
2	uml:Property	Property2_id	protected	Device_id	association xmi:idref=Association_id

memberEnd (2)	xmi:idr...
1	Property1_id
2	Property2_id

Librerías donde está el código generado	
packagedElement	
xmi:type	uml:Artifact
xmi:id	Artifact_id
name	RequiredLibrary
Middleware de comunicaciones	
packagedElement	
xmi:type	uml:ExecutionEnvironment
xmi:id	EE_id
name	CommunicationMiddleware

En la Figura 4 se ilustra el léxico y sintaxis que siguen los modelos TARMAN.xml. Toda TArea Robótica MANipuladora además de su identificador y plataforma MW se caracteriza por una secuencia de componentes y conectores. Los componentes tienen como características la funcionalidad que encapsulan (i.e. el código atómico), la localización de dicho código y el periodo de ejecución. En caso de requerir información de otro componente tendrán puertos de entrada; asimismo, para proporcionar información al exterior tendrá puertos de salida. Los conectores se caracterizan por los extremos de comunicación (fuente y destino) así como el tipo de información transmitida. Para caracterizar los extremos de comunicación será necesario disponer del método correspondiente a la clase atómica que permita actualizar el dato (en el caso de Destino) u obtener su valor (en el caso de Fuente).

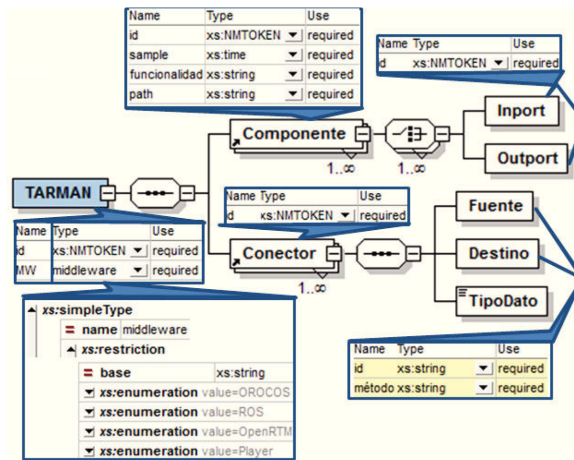

Figura 4: Meta-modelo TARMAN

Hay que destacar que la generación del fichero *TARMAN.xml* es totalmente transparente al usuario. Para realizar dicha transformación, dado que tanto el modelo de entrada como el de salida tiene notación de marcado, la tecnología seleccionada ha sido XML Stylesheet ya que permite filtrar y procesar ficheros XML a través de una serie de plantillas (Tidwell, 2008). La siguiente tabla resume las reglas de transformación principales definidas.

Tabla 6: Reglas de transformación XMI2TARMAN

UML + Perfil TARMAN	TARMAN ML
Componente realizado por una clase estereotipada	Componente
Puerto y uso de una *Interfaz*	Inport
Puerto y realización de una *Interfaz*	Outport
Interfaz + operaciones <get>/<set>	Conector, su fuente y destino. El tipo de dato en función del tipo de parámetro de los métodos

4.2. *Características de los Middleware de comunicación para plataformas robóticas manipuladoras*

Para poder llevar a cabo las transformaciones del M2T, además de tener conocimiento de la estructura e información del modelo de entrada, es también necesario conocer las particularidades de los MW de comunicación. La sección 2.1 de (Azamat et al., 2011) cita las características más relevantes de los MW de comunicación más aceptados en la comunidad científica de la Robótica. Todos los MW salvo ROS están basados en componentes. La Tabla 7 resume las características principales de cada uno de ellos.

De entre los MW basados en componentes, este trabajo se centra en OROCOS debido a su soporte para aplicaciones RT. En este tipo de MW las aplicaciones se forman por un conjunto de

componentes interconectados (Marina et al., 2013). La comunicación entre los componentes se puede llevar a cabo siguiendo el modelo *publicista/suscriptor* para lo cual intervienen los puertos de entrada y salida de los componentes. Por otro lado, también soporta el modelo de comunicación *cliente/servidor* realizado través de las operaciones de petición y respuesta.

Este trabajo también analiza el MW ROS que aunque no está basado en componentes, también está formado por unidades modulares de programación llamadas *nodos*. En este caso, la comunicación entre los nodos también 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.

Tabla 7: características principales de los MW más utilizados en Robótica

MW Basados en Componentes				
OROCOS (Bruyninckx, 2001). Middleware modular que proporciona una serie de librerías entre las que destaca Real-Time Toolkit (RTT) ya que facilita los recursos para poder desarrollar aplicaciones RT				
Comando (Asín.)	Método (Síncr.)	Datos/ Buffer	Parámetros Modificables	Estado
Operación	Operación	Puerto	Propiedad	Preop, stop, run
OpenRTM (2015). Middleware de código abierto desarrollado por AIST				
Comando (Asín.)	Método (Síncr.)	Datos/ Buffer	Parámetros Modificables	Estado
----	Puerto Servicio	Puerto de datos	Interfaz Config.	Created, inactivo, activo
Player (Gerkey et al., 2003). MW desarrollado por la Universidad Southen California.				
Comando (Asín.)	Método (Síncr.)	Datos/ Buffer	Parámetros Modificables	Estado
Comando	----	Interfa z Datos	Puerto Servicio	----
MW Orientado a Nodos				
ROS (Jesse and Ronald, 2012). Conjunto de librerías de código y herramientas de código abierto para ayudar en el desarrollo de aplicaciones robóticas.				
Comando (Asín.)	Método (Síncr.)	Datos/ Buffer	Parámetros Modificables	Estado
----	Servicio	Tópic o	Parámetro	

4.3. M2T – Transformación TARMAN a Código fuente OROCOS

Todo componente OROCOS sigue una misma estructura, fijada en la tarea *TaskContext* proporcionada por la librería RTT. Por otro lado, la lógica de la aplicación que se ejecute en OROCOS se detalla en un fichero de despliegue del MW, siguiendo una estructura determinada en XML (Orocos - the deployment component, 2012).

La Figura 5 ilustra las plantillas para generar la cabecera y el código fuente de los componentes OROCOS de una aplicación.

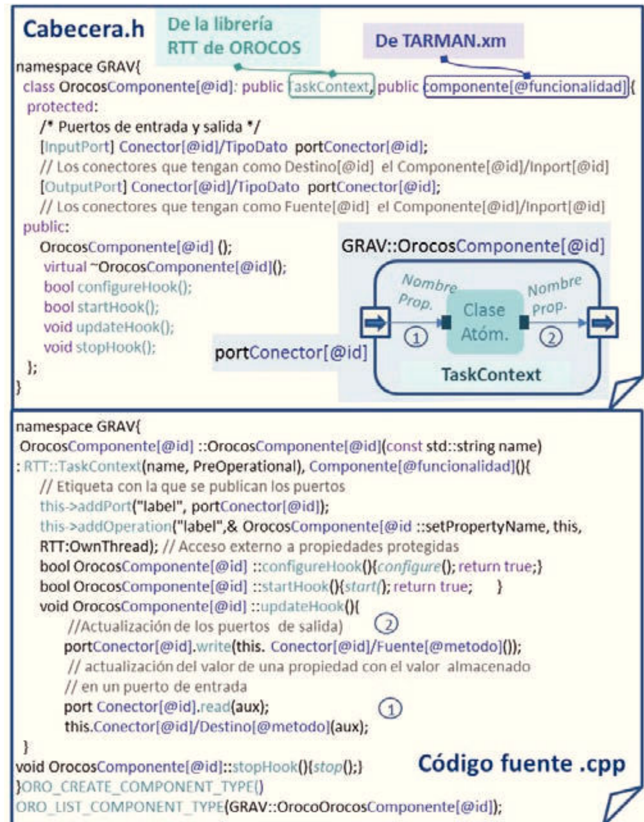

Figura 5*:* Plantillas para la generación de cabecera y código fuente de los componentes OROCOS

Las características de planificación de cada componente se detallan en el fichero de despliegue que es el motor de ejecución de OROCOS ya que recoge la información necesaria para ejecutar las aplicaciones de forma correcta. Tal y como ilustra la Figura 6.a, todo fichero de despliegue compone de tres partes principales (1) librería(s) donde se encuentran los componentes OROCOS de la aplicación, (2) información intercambiada entre los componentes (puntos de conexión) y (3) la planificación de los componentes OROCOS que forman la aplicación. En la Figura 6.b se detalla la información intercambiable entre los componentes que forman la aplicación ejemplo, e.g. *visionValueConn* es un punto de conexión que almacena la información capturada por una cámara.

Para la planificación, OROCOS maneja dos primitivas: ORO_SCHED_RT para planificadores de tiempo real y ORO_SCHED_OTHERS para el resto de casos. La Figura 6.c detalla la información de planificación para un componente cámara. Se trata de una actividad periódica donde la cámara proporciona una imagen cada 33ms.

Las principales reglas de transformación para la generación de código para cualquier MW basado en componentes, tomando como referencia a OROCOS son:

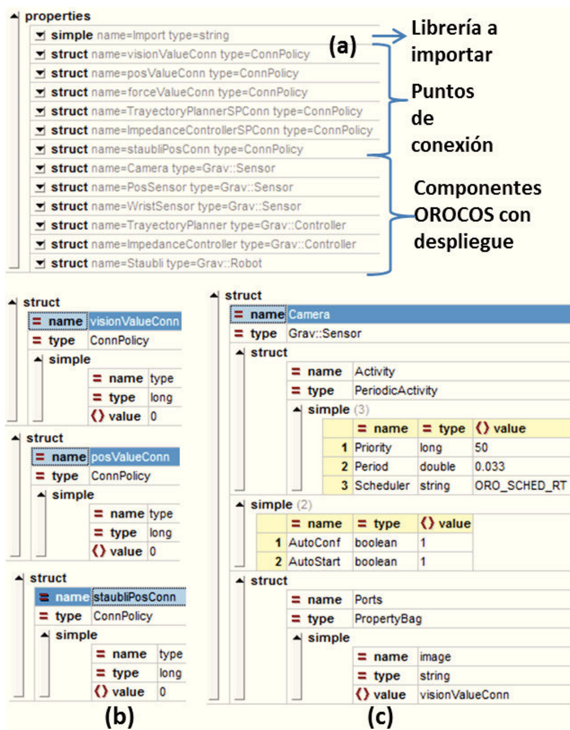

Figura 6*:* Ejemplo de fichero de despliegue. (a) vista general, (b) puntos de conexión y (c) despliegue de un componente OROCOS

- Regla 1: Generación de los componentes de aplicación del MW (MW_App_Comp). Se genera uno por cada componente TARMAN. La interfaz de los MW_App_Comp viene fijado por el motor de ejecución del MW seleccionado. Para el caso de OROCOS dicha interfaz viene fijada por *Task_Context (*texto en verde de la Figura 5*).*
- Regla 2: Proporcionar acceso externo – Puertos de salida. 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.
  Por cada dato puerto se define:
  [OutputPort] *conector[@id]/TipoDato* port*Conector[@id]*;
  Además en el método updateHook se actualiza la información disponible para el exterior:
  portConector[@id].write
  (this.*conector[@id]/Fuente[@metodo]*();
- Regla 3: Proporcionar acceso externo – Puertos de entrada. 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. Por cada dato, se define:
  [OutputPort] *conector[@id]/TipoDato* port*Conector[@id]*;
  Además en el método updateHook se actualiza la información disponible para el exterior:
  portConector[@id].read(aux);
  this.*conector[@id]/Destino[@metodo]*(aux);
- Regla 4: Generación del fichero de despliegue (Figura 6).
  - Localización de las librerías: se obtiene procesando el *path* de los componentes TARMAN.
  - Los puntos de conexión, están directamente relacionados con los conectores TARMAN.
  - La parte correspondiente al despliegue de MW_App_Comp: el periodo se obtiene del *sample* del componente TARMAN. Además, se definirán tantas estructuras puerto como puertos tenga el componente TARMAN.

La implementación de dichas reglas de transformación se ha llevado a cabo nuevamente a través de XSLT. De esta manera se ha definido una XSLT con nueve plantillas que procesa el fichero *TARMAN.xml* y genera tantos MW_App_Comp como componentes tiene la lógica de la aplicación diseñada y también genera el fichero de despliegue.

4.4. M2T – Transformación TARMAN a Código Fuente ROS

Este apartado se centra en la generación de código para que la aplicación diseñada pueda ejecutarse sobre ROS. Para ello se ha hecho nuevamente uso de la tecnología XML stylesheet. En la Figura 7 se presenta a modo de plantilla el procesamiento a seguir para la generación de código de los nodos ROS. Al igual que en el apartado anterior, en itálica aparece resaltada la información necesaria del fichero *TARMAN.xml* para generar la cabecera y funcionalidad del nodo. En este trabajo el código de los nodos estará en C++. Al contrario que sucede en OROCOS, ROS no fija una estructura de nodo.

A continuación se listan las principales reglas de transformación identificadas:

- 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 (véase cabera de Figura 7):
  - Un dato protegido de tipo publicador.
    Si es de tipo básico (ROS msg, 2015):
    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]*();

  En la parte inferior de la Figura 7 se observa que 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 (véase cabera de Figura 7):
  - 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 iCv*Conector[@id]*;

- Método responsable de suscribirse al tópico:
  void *Conector[@id]*CallBack (const TipoDato::ConstPtr& message);

En la parte inferior de la Figura 7 se observa que 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.

```
#include <ros/ros.h>
//includes de tipos de mensajes por cada TipoDato de los conectores
#include "Componente[@funcionalidad].h"
class RosNodeComponente[@id]  : public Componente[@funcionalidad]{
  protected:
      ros::Handle node; //manejador del nodo
      /**             Publicista(s)                   **/
      /*  Por cada puerto de salida buscar aquellos  conectores que tengan como
         Fuente[@id]  el Componente[@id]/Outport[@id]*/
        ros::Publisher msgConector[@id];  // si el tipo de dato es básico
         // si el tipo de dato es una imagen
         image_transport::ImageTransport  iTranspConector[@id];
          image_transport::Publisher iPubConector[@id];
      /**              Suscriptor)                    **/
      /*  Por cada puerto de entrada buscar aquellos  conectores que tengan como
         Destino[@id]  el Componente[@id]/Inport[@id]*/
         ros:: Suscriber msgConector[@id];  // si el tipo de dato es básico
         // si el tipo de dato es una imagen
         image_transport::ImageTransport  iTranspConector[@id];
         image_transport::Publisher iPubConector[@id];
         cv_bridge::CvImagePtr iCv Conector[@id];
  public:
      RosNodeComponente[@id] ();
      virtual RosNodeComponente[@id]();
      // por cada dato al que se suscribe
      void Connector[@id]CallBack(const TipoDato::ConstPtr& message);
      void PublishConector[@id](); // por cada dato que publica
};
```

Cabecera.h

```
#include "RosNodeComponente[@id].h"
RosNodeComponente[@id]::RosNodeComponente[@id](ros::NodeHandle n_): node(n_){
/** por cada dato a publicar **/
    msgConector[@id]=node.advertise<TipoDato>(conector[@id],1); // si es tipo básico
    // si es una imagen
    iPubConector[@id] = iTranspConector[@id].advertise(Conector[@id],1);
/** por cada dato a suscribir **/
    msgConector[@id]=node.subscribe(conector[@id],1,&RosNodeComponente[@id]::
                              Conector[@id]CallBack, this); // si es dato básico
}
/* acción de suscribirse a un tópico. Tantas como tópicos a suscribirse */
RosNodeComponente[@id]::Conector[@id]CallBack(const TipoDato::ConstPtr& message){
 Conector[@id]/Destino[@metodo](message->data); //si es dato básico
}
/* acción de publicar un tópico. Tantas como tópicos a publicar */
RosNodeComponente[@id]::PublishConector[@id](){
// si es tipo básico
    TipoDato message;
    message.data=Conector[@id]/Fuente[@metodo]();
    msgConector[@id].publish(message);
// si es de tipo imagen
    cv_bridge::CvImagePtr message;
     message->image= Conector[@id]/Fuente[@metodo);
    iPubVisionValueConn.publish(message->toImageMsg());
}
/* función principal*/
int main (int argc, char** argv){
    ros::init(argc,argv, RosNodeComponent[@id]);
    ros::NodeHandle n;
    RosNodeComponente[@id] rosNode(n);
    ros::Rate loop_rate(Componente[@id]/@sample);
    while(ros::ok()){
        rosNode.Update();
        rosNode.PublishConector[@id](); // por cada tópico a publicar
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

Código_fuente.cpp

Figura 7: Plantillas para la generación de cabecera y código fuente de los Nodos ROS

- 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 (Aaron and Enrique, 2013).
  - 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 (Aaron and Enrique, 2013) 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 8*:* Ejemplo de fichero launch

5. Casos de Estudio: robots industriales vs servicio

Los robots se pueden clasificar en dos categorías dependiendo de su finalidad y necesidades del mercado para las que hayan sido diseñados. Esta sección presenta dos ejemplos de robótica manipuladora: industrial y de servicios. El primero de ellos está centrado en la resolución de problemas reales industriales trabajando en entornos estructurados. En el segundo caso se

centra en la resolución de tareas en entornos humanizados, no estructurados y desconocidos. Se han elegido estos dos ejemplos tan diferentes para resaltar la reutilización del código independientemente de la tarea, robot y plataforma utilizada. Asimismo, cada caso de uso correrá sobre un MW diferente, el primero de ellos sobre OROCOS y el segundo sobre ROS.

5.1. Aplicación industrial: ensamblaje de faro de vehículos

Hoy en día los faros de vehículos son dispositivos altamente sofisticados que deben pasar un gran número de inspecciones de calidad, demandadas por los fabricantes (Javier et al., 2012) (Satorres-Martínez et al., 2013). Una de las etapas del proceso de producción de los faros consiste en el ensamblaje de los componentes, donde la acción principal es el posicionamiento y posterior fijado de la lente -hecha de policarbonato- sobre la estructura base hecha de polipropileno (Figura 9).

El proceso de fijado de la lente consiste en mover la lente dentro del canal -con un ancho de unos 2 mm- usando el robot para reducir al máximo las fuerzas de contacto. Se han usado dos sensores: un sensor de fuerza acoplado a la muñeca del brazo robótico (*Atilsa*, Figura 10.a), que determina las fuerzas y el par generado por el manipulador con el entorno; y un sensor de visión (Guppy80, Figura 10.a) cuya misión es, usando algoritmos de procesado de imágenes, localizar la posición del canal de pegado de forma precisa.

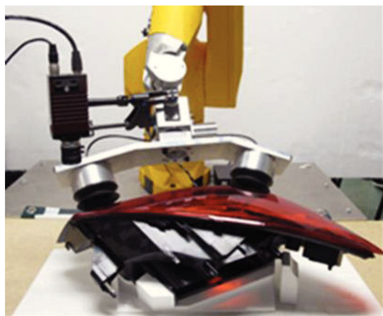

Figura 9*:* fotografía del ensamblaje de un faro

La lógica de dicha aplicación se ilustra en la Figura 10.a con el diagrama de componentes UML. El software que maneja la cámara Guppy80 proporciona una imagen cada 33milisegundos. El componente *PosSensor*, realizado por el algoritmo cuatro puntos, a partir de dicha imagen genera la posición del target.

Por otro lado, el software que maneja el sensor de fuerza Atilsa (*WristSensor*) proporciona cada 0.4milisegundos el par generado por el manipulador con el entorno. El generador de trayectorias, una vez que conoce la posición actual del robot y la posición del target, genera la trayectoria a seguir por el robot. El control de impedancia requiere de la trayectoria generada por el generador de trayectorias, el par de fuerza y la posición actual del robot para calcular la siguiente posición.

Así mismo, la plataforma HW/SW se detalla en Figura 10b. Se disponen de tres PCs con el MW de OROCOS, el código del repositorio (Library) y además se indica qué componentes se despliegan en qué PCs.

El generador de código procesa el fichero XMI y genera los componentes OROCOS de aplicación y el fichero de despliegue correspondientes.

A modo de ejemplo, la Figura 11 muestra el fichero TARMAN.xml para dicha aplicación. Como se ha comentado anteriormente el usuario no tendría conocimiento alguno de dicho documento. La Figura 12 ilustra el código generado para el componente OROCOS Camera.

Todos los componentes generados serán compilados y almacenados en una librería. El fichero de despliegue resultante para esta aplicación se ilustra en la Figura 6.

Finalmente, la aplicación en tiempo Real se arranca sobre el middleware OROCOS con el comando *deployer-xenomai -s deploymentComponentFileName.xml*.

5.2. Aplicación de servicios: 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.

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, bien con la cinta transportadora o entre ellos mismos.

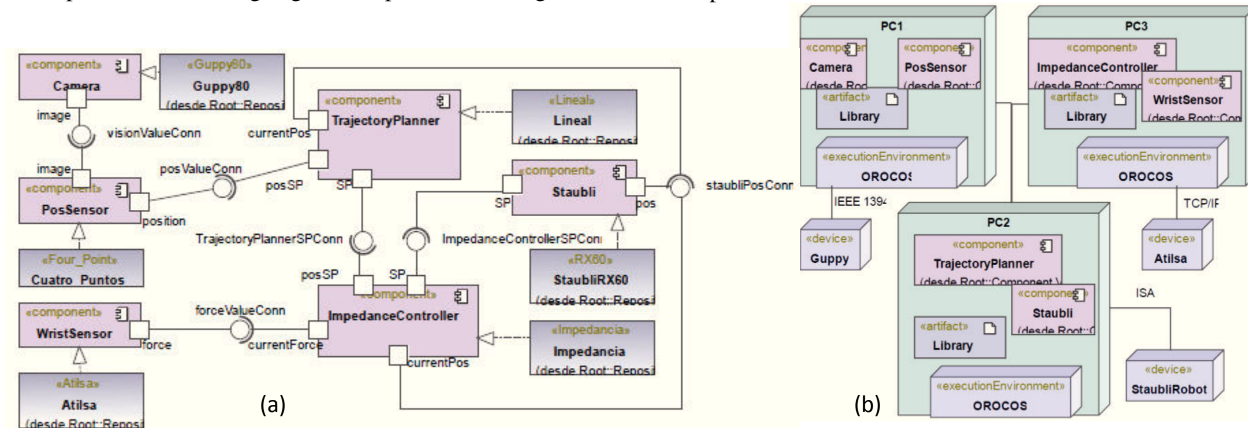

Figura 10*:* Modelado del montaje de faros de vehículos: (a) funcionalidad y (b) plataforma HW/SW

TARMAN

id	Faros
MW	OROCOS

Componente (6)

	id	sa...	funcion...	path	Inport	Outport
1	Camera	00:33:00	Guppy80	repositorio		Outport (1): id — 1 image
2	PosSensor	00:00:00	CuatroPuntos	repositorio	Inport (1)	Outport (1)
3	TrajectoryPlanner	00:00:00	Lineal	repositorio	Inport (2)	Outport (1)
4	ImpedanceControlle r	00:00:00	Impedancia	repositorio	Inport (2): id — 1 posSP, 2 currentForce	Outport (2): id — 1 SP, 2 currentPos
5	Staubli	00:00:40	StaubliRX60	repositorio	Inport (1)	Outport (1)
6	WristSensors	00:00:40	Atilsa	repositorio		Outport (1)

Conector (6)

	id	Fuente	Destino	Tipo...
1	visionValueConn	Fuente id=Camera/image...	Destino (1)	CV:Mat
2	posValueConn	Fuente id=PosSensor/po...	Destino (1)	std::vector <double>
3	forceValueConn	Fuente id=WristSensors...	Destino (1)	std::vector <double>
4	TrajectoryPlannerSP Conn	Fuente: id TrajectoryP lanner/SP; método getSP	Destino (1): id / método — 1 ImpedanceControl ler/posSP, setPosSP	std::vector <double>
5	ImpedanceController SPConn	Fuente id=ImpedanceCo...	Destino (1)	std::vector <double>
6	staubliPosConn	Fuente: id Staubli/pos; método getPos	Destino (2): id / método — 1 TrajectoryPlanner /currentPos, setCurrentPos; 2 ImpedanceControl ler/currentPos, setCurrentPos	std::vector <double>

Figura 11*:* Fichero TARMAN de la aplicación

```
namespace GRAV{
OrocosCamera::OrocosCamera(const std::string name): RTT::TaskContext(name,
PreOperational),Guppy80(){
  this->addPort("image", portImageRaw);
  this->addOperation("setWidth",&OrocosCamera::setWidth, this, RTT:OwnThread);
  this->addOperation("getWidth",&OrocosCamera::getWidth, this, RTT:OwnThread);
  this->addOperation("setHeight",&OrocosCamera::setHeight, this, RTT:OwnThread);
  this->addOperation("getHeight",&OrocosCamera::getHeight, this, RTT:OwnThread);
  this->addProperty("ip",ip);
}
bool OrocosCamera::configureHook(){configure(); return true;}
bool OrocosCamera::startHook(){start(); return true;}
void OrocosCamera::updateHook(){
  portImageRaw.write(this.getImageRaw());}
void OrocosCamera::stopHook(){stop();}
}ORO_CREATE_COMPONENT_TYPE()
ORO_LIST_COMPONENT_TYPE(GRAV::OrocosCamera);
```

Figura 12*:* Código fuente del componente OROCOS Camera

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.

La Figura 13 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 14 ilustra la funcionalidad de la aplicación así como la arquitectura hardware. En este caso la aplicación correrá sobre el MW de ROS. La Figura 15 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 8).

Figura 13: seguimiento de un objeto en movimiento por parte del robot Meka

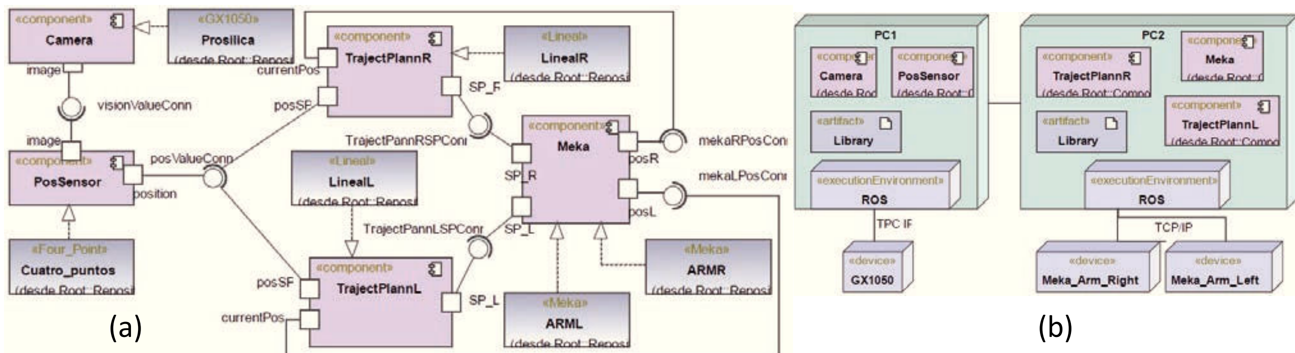

Figura 14: Modelado de la tarea de seguimiento de un objeto en movimiento: (a) funcionalidad y (b) plataforma HW/SW

```
#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 15: Código Fuente del nodo ROS cámara

Finalmente, es importante destacar que parte del software atómico del caso de estudio 1 se ha reutilizado en el caso de estudio 2, aun siendo plataformas diferentes. Concretamente, como se puede apreciar en la Figura 10.a y Figura 14.a tienen como software atómico común el algoritmo de 4 puntos que genera una posición ante una imagen facilitada. Además, también comparten código en lo referente al algoritmo de generación de trayectorias para el staübli en la aplicación industrial, así como la generación de trayectorias para el brazo derecho e izquierdo del robot Meka en la aplicación de servicios.

6. Conclusiones

Este trabajo presenta una plataforma para dar soporte al ciclo de desarrollo a aplicaciones robóticas manipuladoras 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 de una manera independiente a la plataforma de ejecución y además haciendo uso de una notación estándar. Se ha definido el perfil UML TARMAN que puede ser utilizado por cualquier herramienta de modelado UML que soporte el estándar XMI. Además, gracias al módulo cpp2xmi es posible importar a la herramienta de modelado la interfaz de aquellos módulos de software atómico necesarios. El modelado de la lógica y arquitectura de las aplicaciones se realiza con notación UML, lo cual hace que la metodología propuesta sea genérica y válida a cualquier herramienta de modelado UML que soporte el estándar XMI.

Para la generación de código se han seguido las directrices M2M y M2T de MDE. En concreto, se ha definido una transformación M2M que tiene como modelo de entrada el fichero XMI, que será procesado para generar el modelo *TARMAN.xml*. Este procesado únicamente tiene en cuenta aquellos componentes UML que estén realizados por una clase que represente el interfaz del código atómico que encapsula, i.e. dichas clases tendrán asignado algún estereotipo del perfil *TARMAN_profile*. De esta forma el fichero *TARMAN.xml* tendrá la mínima información necesaria para la generación de código y es el modelo de entrada para el segundo tipo de transformación.
En este trabajo se han presentado dos ejemplos de M2T ambos partiendo del mismo modelo de entrada. El primero de ellos es válido para los MW de comunicaciones basados en componentes, tomando como ejemplo OROCOS. El segundo para ROS. De la misma manera, los autores, han presentado dos casos de estudio uno para robótica manipuladora industrial y otro para robótica manipuladora de servicios. Con dichos casos de estudio se ha demostrado la reutilización de código atómico para diferentes tareas así como la generación automática de código que se ejecute en diferentes MW de comunicaciones.

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. Por otro lado, esto también permitirá a los autores definir como trabajo futuro una herramienta de modelado que ofrezca una interfaz más amigable a los usuarios finales, ofreciendo un léxico y sintaxis más afín a su dominio y evitar así el tener que manejar los diagramas de componentes y de despliegue UML. Esta herramienta manejará internamente, y de forma transparente al usuario el modelo *TARMAN.xml*. Asimismo, tendrá incluida la generación de código final M2T.

English Summary

An UML based approach for designing and coding automatically robotic arm platforms.

Abstract

Today, robotics manipulator is a crucial discipline in modern production industrial facilities and in a near future; it will also be decisive in the human quotidian society. Consequently, currently there is a growing demand of applications with arm-based robots with requirements such as: reutilization, flexibility and adaptability. Unfortunately, there is a lack of standardization of hardware and software platforms, so the satisfaction of these requirements is too difficult. In this sense, there is a necessity of a methodology that guides along application design, implementation as well as the execution of the software systems. This work, explores the advantages of Model Driven Engineering (MDE) for the design and development of applications performed by manipulator robots. In fact, an UML based approach is proposed that supports the design of robotic tasks and an automatic code generation for the most spread robotic communication Middlewares has been also developed. More specifically, the target code generation for OROCOS and ROS communication Middlewares has been detailed. Finally, two case

studies have been presented one for industrial field and the other for service sector. The former runs on OROCOS and the latter on ROS.

Keywords:

Manipulator robots, UML-Unified Modeling Language, MDE-Model Driven Engineering, ROS-Robotic Operating System, OROCOS- Open Robot Control Software.

Agradecimientos

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

Referencias

Aaron Martínez, Enrique Fernández, 2013. Learning ROS for Robotics Programming, Packt Publishing ltd.
Alonso D., Vicente-Chicote C., Ortiz F., Pastor J., Álvarez B., 2010. V3CMM: a 3-view component metamodel for model-driven robotic software development. Journal of Software Engineering for Robotics, 3-17.
Aracil Rafael, Balaguer Carlos, Armada Manuel, 2008. Robots de Servicio. Revista Iberoamericana de Automática e Informática Industrial 5(2), 6-13.
Atkinson Colin, Kühne Thomas, 2003. Model-driven development: a metamodeling foundation. IEEE Software 20(5),36–41.
Azamat Shakhimardanov, Jan Paulus, Nico Hochgeschwender, Michael Reckhaus, 2011. Deliverable D-2.1 Best Practice Assessment of Software Technologies for Robotics. [Online] Disponible en: http://www.best-of-robotics.org/pages/publications/BRICS_Deliverable_D2.1.pdf
Balasubramanian K., Gokhale A., Karsai G., Sztipanovits J., Neema S., 2006. Developing applications using model-driven design environments. Computer 39(2), 33–40.
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 3(2), 79-89
Barner S., Geisinger M., Buckl C., Knoll A., 2008. EasyLab: model-based development ofsoftware for mechatronic systems. Proc. IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications, 540 –545.
Bischoff R., Guhl T., Prassler E., Nowak W., Kraetzschmar G., Bruyninckx H., Soetens P., Haegele M., Pott A., Breedveld P., Broenink J., Brugali D., Tomatis N., 2010. BRICS – best practice in robotics. Proc. of 41st International Symposium on and 6th German Conference on Robotics (ROBOTIK), 1 –8.
Booch G, Rumbaugh J, Jacobson I (2005) The unified modelling language user guide, 2nd Edition, Addison-Wesley Professional.
Brooks A., Kaupp T., Makarenko A., Williams S., Oreback A., 2005. Towards component-based robotics. Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems, 163 – 168.
Brugali D., Shakhimardanov A., 2010. Component-based robotic engineering (Part II) [Tutorial]. Robotics Automation Magazine, IEEE 17(1), 100 – 112.
Bruyninckx H. , 2001. Open robot control software: The OROCOS. Proc. of IEEE Int. Conf. on Robotics and Automation (ICRA), 2523-2528
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(4), 557 – 573.
Estévez E., Sánchez-García A., Gámez-García J., Gómez-Ortega J., Satorres-Martínez S., 2015. A novel model-driven approach to support development cycle of robotic systems. International. Journal of Advanced Manufacturing Technology, 1-15.
Gabriel J. Garcia, Juan A. Corrales, Jorge Pomares, Fernando Torres, 2009. Survey of Visual and Force/Tactile Control of Robots for Physical Interaction in Spain. Sensors 9, 9689-9733
Garcia H, Bruyninckx H, 2014. Tool Chain (BRIDE) delivered as BRICS software distribution. [online] http://www.best-of-robotics.org/bride/
Gerkey B., Vaughan R., Howard A., 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. Proc. of the International Conference on Advanced Robotics.
Javier Gámez García, Alejandro Sánchez García, Silvia Satorres Martínez, Juan Gómez Ortega, 2012. Ensamblaje automático de piezas con desviaciones dimensionales. Revista Iberoamericana de Automática e Informática Industrial 9(4), 383-392
Jesse Russell, Ronald Cohn, 2012, ROS (Robotic Operating System, VSD
Marina Vallés, Jose I. Cazalilla, Ángel Valera, Vicente Mata, Álvaro Page, 2013. Implementación basada en el middleware OROCOS de controladores dinámicos pasivos para un robot paralelo. Revista Iberoamericana de Automática e Informática industrial 10, 96–103.
Michael Geisinger, Simon Barner, Martin Wojtczyk, Alois Knoll, 2009. A software architecture for model-based programming of robot systems. LNCS, Advances in Robotics Research, Springer, 135–146.
Miller J., Mukerji J., 2003. MDA guide version [Online]. Disponible en: http://www.enterprisearchitecture-info/Images/MDA/
OPENRTM, 2015.
[Online] Website: http://www.openrtm.org/openrtm/en/node/780
Orocos - the deployment component, 2012. [Online]. http://www.orocos.org/stable/documentation/ocl/v2.x/doc-xml/orocos-deployment.html
ROS msg, 2015. [Online] http://wiki.ros.org/msg
ROS sensors_msgs, 2015 [Online] http://wiki.ros.org/sensor_msgs
Sanchez-Garcia A., Estevez E., Gomez Ortega J., Gamez Garcia J., 2013. Component-based modelling for generating robotic arm applications running under OROCOS middleware. IEEE International Conference on Systems, Man, and Cybernetics. 3633-3638.
Satorres-Martínez, S., Gómez-Ortega J., Gámez-García J., Sánchez-García A., Estévez-Estévez E., 2013. An industrial vision system for surface quality inspection of transparent parts. International Journal of Advanced manufacturing Technology 68(5-8), 1123-1136
Schlegel C., Steck A., Brugali D., Knoll A., 2010. Design abstraction and processes in robotics: From code-driven to model-driven engineering. Simulation, Modeling, and Programming for Autonomous Robots, LNCS, Eds. Springer Berlin/Heidelberg 6472, 324–335.
Selic B., 2003. The pragmatics of model-driven development. IEEE Software 20(5):19–25Steinberg D., Budinsky F., Paternostro M., Merks E., 2008. EMF: Eclipse Modeling Framework, 2nd ed. Addison-Wesley Professional
Tidwell, 2008. D. XSLT,2nd Edition, O'REILLY.
Valera A., Juste D., Sánchez A. J., Ricolfe C., Mellado M., Olmos E., 2012. Aplicación de la Arquitectura Orientada a Servicios Universal Plug-and-Play para facilitar la Integración de Robots Industriales en Líneas de Producción. Revista Iberoamericana de Automática e Informática Industrial (9), 24-31.
XML Metadata Interchange Specification, 2014. [Online] Disponible en: http://www.omg.org/spec/XMI/.