The International Journal of Advanced Manufacturing Technology (2018) 97:1195–1207
https://doi.org/10.1007/s00170-018-1976-z

ORIGINAL ARTICLE

ART^{2}ool: a model-driven framework to generate target code for robot handling tasks

E. Estévez[1] · Alejandro Sánchez García[1] · Javier Gámez García[1] · Juan Gómez Ortega[1]

Received: 23 May 2017 / Accepted: 2 April 2018 / Published online: 22 April 2018

Abstract

Nowadays, robotic manipulation tasks are present in modern production industries, making robotics a decisive discipline in the industrial sector. Additionally, in a short period of time, handle robots will be also become essential in daily life. There is an increase in demand for applications for handle robots with software requirements such as reusability, flexibility, and adaptability. Unfortunately, the current lack of standardization of hardware and software platforms hinders the fulfillment of these requirements. Hence, it is necessary to define a methodology that provides guidelines to design, implement, and support at runtime of such types of applications. This work explores the advantages of Model Driven Engineering (MDE) in the design and development of tasks performed by handle robots. Concretely, the authors present the ART^{2}ool (Arm based Robotic Tasks modeling Tool), a MDE framework, which is very useful for application domain experts, because it guides them along the design of the application functionality, abstracting from the emerging techniques. Besides, the proposed framework supports an automatic code generation by Mode to Text transformation techniques for component-based and ROS communication middleware, achieving the requirements mentioned previously.

Keywords Handle robots · Model Driven Engineering · ROS—Robotic Operating System · OROCOS

1 Introduction

Nowadays, robotic manipulation is a decisive discipline in manufacturing and service industries. In fact, there is an increase in investment by public institutions in these industries in order to encourage innovation, economic growth, and job creation. Hence, initiatives such as [1–4] promote the reusability, integration, flexibility, and optimization of industry processes, which are the major requirements demanded by modern production facilities due to the continuous changing market demands.

Unfortunately, the fulfillment of previous commented requirements in handle robot-based applications is a very complex issue because of (1) the great variability of robots in the market for different purposes and execution platforms, (2) the proprietary solutions due to missing standards, and (3) the lack of interoperability between those tools involved in the development cycle of the tasks performed by robots in production processes.

Independently of the task, every handling robot application is composed of distributed, heterogeneous software components (i.e., sensors, processing algorithms, and controllers) interacting in a highly dynamic, uncertain environment. Nevertheless, the integration and collaboration among these modules are not easy because of the lack of standards. In fact, although many elements, such as sensors, actuators, auxiliary elements, and tools, need to be added to a robot to make it more flexible and adaptable, their integration and collaboration are not easy because the followed software development methodology does not keep reusability in mind.

In order to ensure meeting these requirements, hardware and software platforms should allow developers to cope with complexity imposed by applications themselves: hardware, software, time requirements, and distributed computing environments. In this context, this work explores the advantages delivered by the use of Model Driven Engineering (MDE) [5] to provide support to the development cycle of applications based on handling robots.

In recent times, the MDE discipline is being introduced in the robotics field [6, 7]. Hence, for instance, [8] shows the

✉ E. Estévez
eestevez@ujaen.es

[1] Departamento de Ingeniería de Electrónica Automática, EPS de Jaén, Jaén, Spain

EasyLab tool for mechatronic systems. Later, the authors adapted this tool to the Robotino Mobile Robot© platform [9]. This is based on two graphical proprietary languages: Synchronous Data Model for software and other language for hardware description as a collection of sensors and actuators. The former is very similar to the Sequential Function Chart of IEC 61131-3 standard [10], but the latter, as far as authors know, does not follow any wide spread standard. [11] by means of a set of Unified Modeling Language diagrams defines robotic applications, but also generates ADA code, running over CORBA. More recently, [12] shows the MDSD toolchain for the SmartSoft framework which provides a stringent component model to define middleware independent components and their interfaces. Currently, two implementations of the SmartSoft are available. The first is based on CORBA (using ACE/TAO library), and the other is based on simple message passing (using Adaptive Communication Environment).

Following MDE techniques, The Best Practices in Robotics (BRICS) European Project defines the BRIDE (BRICS Development Environment) toolchain [13] in order to facilitate the design of robotic applications. BRIDE is based on Eclipse Modeling Framework (EMF) [14]. Before starting the design of applications, the target middleware must be selected. As far as authors know, BRIDE toolchain provides customized support for two platforms: OROCOS [15] and ROS (Robotic Operating System) [16]. Thus, depending on the selection, the toolchain allows interconnecting OROCOS components or ROS nodes [17] graphically. In this context, BRIDE allows the reuse of OROCOS application components or ROS nodes in different handling robot tasks.

ReApp [18] is a component-based modeling tool for ROS and generic IOs for FANUC robots which allows the reuse and replace of ROS nodes without the necessity of redefinition of the robotic task.

Even though previously cited works use Model Driven Design techniques, the reusability is achieved in different handling robot applications running over same middleware. Nevertheless, the required code that performs the configuration of a sensor and the measurement, or that performs the control strategies, it could not be reused in the case of having the same sensor(s)/actuator(s) or control strategies in different handling robotic tasks running in different middlewares.

This work goes one step further, it proposes the ART^{2}ool (Arm based Robotic Tasks modeling Tool) framework that guarantees code reuse independently of the runtime platform. It provides the mechanisms for defining graphically the functionality of the applications, making use of concepts managed by application domain experts (i.e., sensors, processing modules, and actuators), later lists the runtime platform (middleware) to select in which one is going to be running the robotic task. Finally, this framework has a code generator to generate the target code according to selected runtime platform properties.

A previous work of the authors [19] identified and characterized common interfaces for the components that can appear in every handle robot-based application, independently of the manufacturer in case of sensors and actuators.

The reuse of this atomic code is guaranteed with the ART^{2}ool framework. Because it is composed by two main modules: (1) a graphical editor, designed specifically for robotic field experts. This editor is essentially twofold, on one hand, it offers guidelines for defining the functionality of the tasks but it also abstracts the domain experts from underlined technologies. (2) Code generator module that applying Model to Text transformation techniques of MDE which, once selected the runtime platform, generates automatically the target code ready to be run over runtime platform. Currently, this code generator supports the generation for OROCOS and ROS middlewares.

The remainder of this work is as follows: Sect. 2 describes the guidelines for modeling handle robot-based applications. This section concludes with a meta-model that describes the lexicon and syntax of applications functionality. Section 3 identifies the transformation rules for generating target code for the most wide spread middleware in the robotics community. Section 4 details ART^{2}ool framework. The proposed framework is tested in Sect. 5 with a service robot case study. Finally, Sect. 6 introduces conclusions of this paper.

2 Modeling of handle robot-based applications

Previous works like [20, 21] have identified and characterized which types of components take part in handle robot-based applications. To be more precise, there are three types of components: sensors, actuators, and processing algorithms such as trajectory planning.

Figure 1, with a Unified Modeling Language (UML) class diagram, shows the common characteristics of all identified types of components. *Atomic Task* is an abstract UML class that collects the minimum common information of all types of components that take part in handling robotic applications. It is formed by sample property whose value indicates how many times the component is executed per second (*sample*). Furthermore, every component could be configured, started, and stopped. All components (sensors, robots, and controllers) inherit an essential property from the Atomic Task.

Hence, all type of sensors inherit sample property which contains information about how often the measurement must be updated but also the are defined by the nature of the magnitude to be measured (*type*), the number of samples (*size*), and the *value* of the measurement. Furthermore, there are properties and methods which are dependent on the magnitude's nature. Consequently, for capturing images, a raw

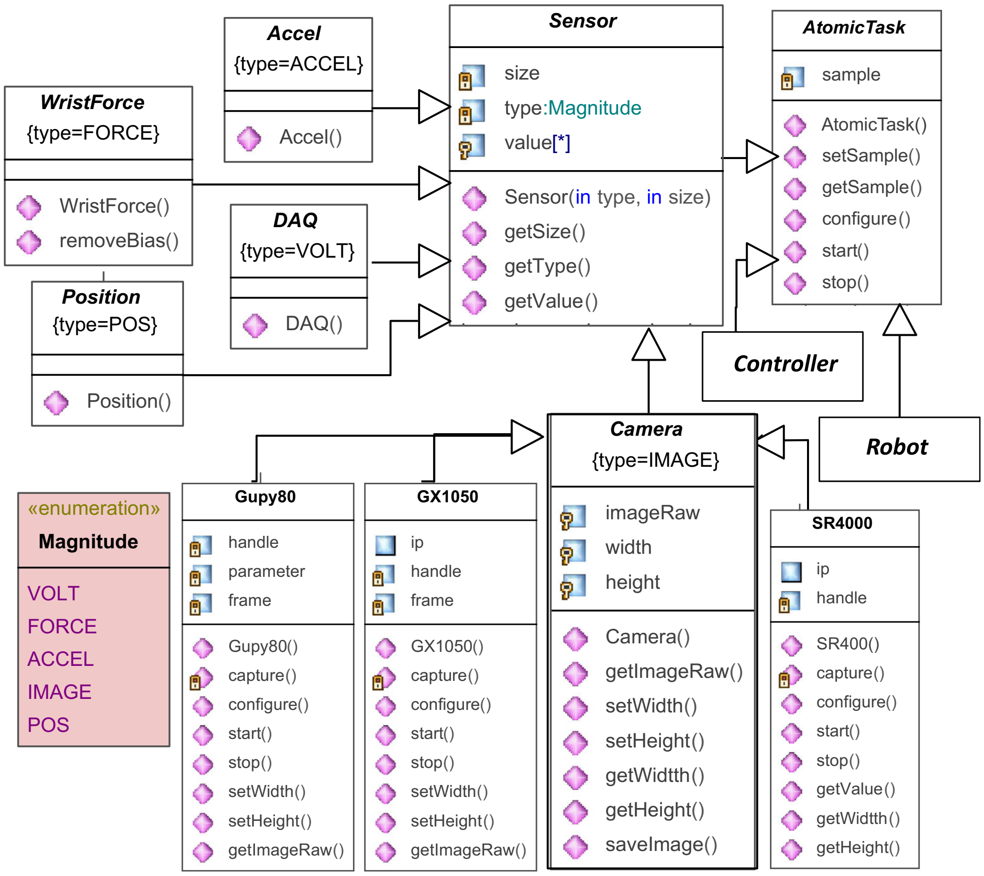


Fig. 1 Characterization of the atomic code components of handle robot-based applications

image format with its width and height is added jointly with the corresponding *set* and *get* methods. This abstract class is the starting point capture images for all cameras. Then, for a specific camera (e.g., Guppy F-080C, Prosilica GX1050), manufacturer-dependent parameters (as private properties) and the *capture* private method are added (See Guppy80, GX1050, and SR4000 Classes of Fig. 1 that represent the interface of the atomic code for managing the corresponding cameras independently of the logic of the application.).

Manufacturer-independent proposed interfaces help developers to add new minimal codification units in a database. On the other hand, they provide a complete abstraction to application domain experts. Hence, robotics domain experts do not require knowledge about manufacturer proprietary drivers [19].

The logic of all handle robot-based applications can be defined as a set of interconnected components [7]. Every component will encapsulate a minimal codification unit to which the logic of the application is added. This is possible by defining an external accessibility to the component by means of input and output ports. The dialog between components is specified by a connector composed by a set of connections which represent a single data interchange.

The following figure shows a meta-model highlighting only those properties which relate the logic to the isolated atomic source code units.

Hence, every component-encapsulated atomic code is characterized by two properties: identifier of such a code (*refAtomicCode*) and the path (*pathAC*). Connectors represent the dialog between components where *source* and *target* properties are required for modeling the sense of such communication. Finally, connections normally will collect the value of a protected property of an atomic code. Therefore, the name of the method to get the value of the source component (*refSourceMet*) and the name of the method to set the value to target component (*refTargetMet*) are required. The instant when the values are updated is related to the value of a sample property of the atomic code (see Fig. 1). So, if this property has a value different from zero, this implies that the value is updated periodically. Otherwise, the value is updated by an event (i.e., this is an on-demand update).

3 Automatic generation of target code for handle robot-based applications

This section is centered in the code generation phase. MDE recommendations have been followed which rely on the model and model transformation concepts in order to automate the software development process. MDE defines two kinds of transformations: Model to Model (M2M) and Model to Text

(M2T), and both have a model as input. In this case, the second type of transformation is defined where the input model is an instance of the Fig. 2 meta-model, and the target code depends on the selected platform. Therefore, the identification and specification of M2T transformation rules implies having knowledge not only of the structure of the input model but also of the particularities of the target platform.

The following sub-sections first detail the main characteristics of the most wide spread communication middleware (MW) in the robotics field and then identify the transformation rules for the most common component-based MW (e.g., OROCOS) and for ROS, which is a node-based platform which provides libraries and tools to help software developers create robot applications.

3.1 Main features of selected communication middleware

The M2T generator module needs knowledge about the target code structure and requirements. Section 2.1 of [22] describes the main features of the most accepted communication middleware in the robotics scientific community. Table 1 summarizes the main characteristics, all except ROS are component-based communication middleware.

Processing the information of the previous table, the authors identified the main transformation rules to generate code for listed runtime platforms:

- R1: Generation of application dependent software units (components/nodes).
- R2: Publish data.
- R3: Subscribe to data.
- R4: Middleware execution engine.

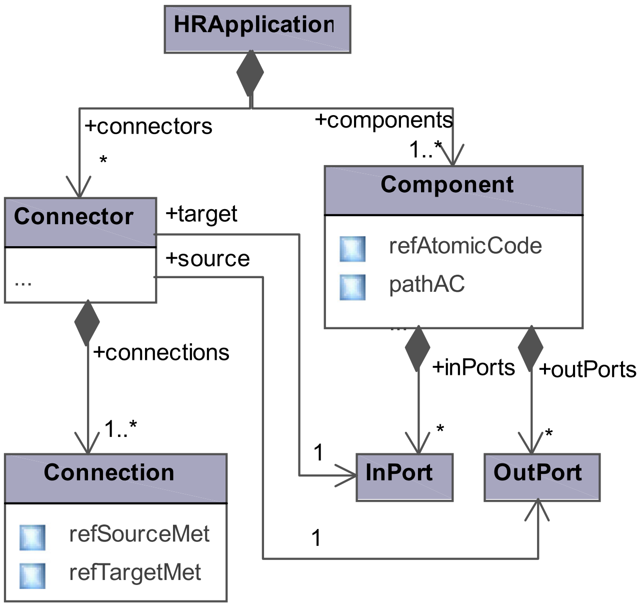


Fig. 2 Meta-model of the logic of the application

This paper selects OROCOS from those component-based MW because of the fact that it supports real-time applications. Every application running over this MW is a set of interconnected OROCOS application components. The communication between those components could be performed following Publish/Subscribe or Client/Server models. The former is achieved by input and output ports, the latter, by means of request and response operations. Hence, Sect. 3.2 of this work details the particularities of R1...R4 in order to generate target code to be run over the OROCOS runtime platform.

In order to demonstrate the code reuse, this paper also considers ROS in which, every application is formed by a set of modular programming units called nodes. The communication between nodes could be performed by Publish/Subscribe and Client/Server communication models as well. The former is achieved with a topic and message interchange, so a node that is interested in making data accessible to other nodes, publishes a topic. In the same way, if a node requires to access specific information must subscribe to the corresponding topic. On the other hand, if the communication model is Client/Server, the node that acts as server remains waiting for a request from the client. When a Client Node makes a request, the Server node performs a processing (service) and responds to client node. Hence, in this case, two messages are interchanged (request and response). Actually, this interaction is presented as a remote procedure call. When many nodes are running, it is convenient to render peer-to-peer communications graphs. Section 3.3 details the particularities of R1...R4 in order to generate target code to be run over the ROS runtime platform.

3.2 Transformation rules for OROCOS

The RTT provides the core of the OROCOS component's interface/structure (*TaskContext* Class), and the logic of the application is collected in a *DeploymentComponent* markup language model.

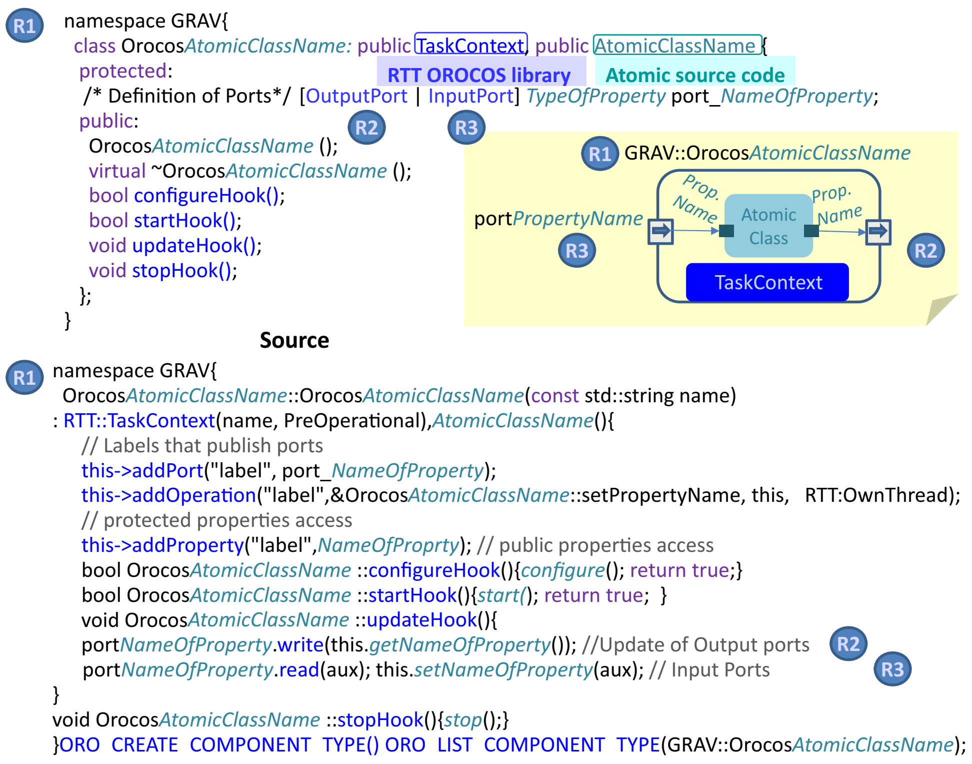
Figure 3 shows the defined templates for generating automatically both header and source files for every application dependent on the OROCOS component. Additionally, scheduling properties of OROCOS components must be detailed in the *DeploymentComponen*t markup language (ML) file, which acts as the execution engine of this MW, because it collects information to ensure a successful execution of applications.

DeploymentComponent file is composed of three main parts: (1) path of the libraries where application OROCOS components are and (2) connection points collecting the information interchanged between components and (3) component execution scheduling. OROCOS manages two primitives for selecting the schedulers: ORO_SCHED_RT for real-time schedulers and ORO_SCHED_OTHERS for the rest types of schedulers.

Table 1 Main features of the MWs most spread in the robotics domain

Component-based middlewares				
OROCOS [15]. Modular MW that provides a set of libraries from which the Real-Time Tookit (RTT) must be mentioned, because it provides the resources for developing real-time applications.				
Command (Asyn.)	Method (Syn.)	Data/buffer	Modifiable parameters	State
Operation	Operation	Port	Property	Preop, stop, run
OpenRTM [23]. Open source middleware developed by AIST.				
Command (Asyn.)	Method (Syn.)	Data/buffer	Modifiable parameters	State
–	Port service	Data port	Interface config.	Created, inactive, active
Player [24]. MW developed by the Southern California University.				
Command (Asyn.)	Method (Syn.)	Data/buffer	Modifiable parameters	State
Command	–	Interface data	Port Service	–
Node-focused middlewares				
ROS [16]. A set of code libraries and open source tools to help with the development of robotics applications.				
Command (Asyn.)	Method (Syn.)	Data/buffer	Modifiable parameters	State
–	Service	Topic	Parameter	

Fig. 3 Templates for generating header and source code of application OROCOS components



```
namespace GRAV{
 class OrocosAtomicClassName: public TaskContext, public AtomicClassName {
  protected:
   /* Definition of Ports*/ [OutputPort | InputPort] TypeOfProperty port_NameOfProperty;
  public:
   OrocosAtomicClassName ();
   virtual ~OrocosAtomicClassName ();
   bool configureHook();
   bool startHook();
   void updateHook();
   void stopHook();
  };
}
```

Source

```
namespace GRAV{
 OrocosAtomicClassName::OrocosAtomicClassName(const std::string name)
: RTT::TaskContext(name, PreOperational),AtomicClassName(){
   // Labels that publish ports
   this->addPort("label", port_NameOfProperty);
   this->addOperation("label",&OrocosAtomicClassName::setPropertyName, this,   RTT:OwnThread);
   // protected properties access
   this->addProperty("label",NameOfProprty); // public properties access
   bool OrocosAtomicClassName ::configureHook(){configure(); return true;}
   bool OrocosAtomicClassName ::startHook(){start(); return true;  }
   void OrocosAtomicClassName ::updateHook(){
   portNameOfProperty.write(this.getNameOfProperty()); //Update of Output ports
   portNameOfProperty.read(aux); this.setNameOfProperty(aux); // Input Ports
}
void OrocosAtomicClassName ::stopHook(){stop();}
}ORO CREATE COMPONENT TYPE() ORO LIST COMPONENT TYPE(GRAV::OrocosAtomicClassName);
```

The following is detailed how implement R1…R4 rules for generating OROCOS application components and the corresponding *DeploymentComponent* file which acts as the engine in OROCOS runtime platform.

- Rule 1: Generation of application dependent components. An OROCOS application component is generated from a component modeled in the logic of the application. As stated previously, the interface of the resulting component follows the structure fixed by the execution engine of the selected MW. For OROCOS, this structure is provided by *Task_Context* (see Fig. 3).
- Rule 2: Publish data. Those output ports (*OutPorts* in Fig. 2) which appear as source in *connector*s are the result of the component providing external accessibility to this data. To publish data in OROCOS, first, an output port is defined: [OutputPort] *typeOfProperty* port_*nameOfProperty*. Later, an updateHook method, provided by Task_Context, is updated with the external accessibility of defined output port: port*NameOfProperty*.write(this.get*NameofProperty*());
- Rule 3: Subscribe to data. Those input ports (InPorts in Fig. 2) which appear as target in connectors are the result of the component subscription to data. To subscribe to data in OROCOS, first an input port is defined: [InputPort] *typeOfProperty* port_*nameOfProperty*. Later, the updateHook method is updated with port*NameOfProperty*.read(aux); this.set*NameOfProperty*(aux);
- Rule 4: Generation of *DeploymentComponent* file. As it is formed by three parts, (1) path property processing, (2) connection points in OROCOS directly related with connectors, and (3) scheduling information of every application, OROCOS components are generated by processing the value of sample property. More specifically, if the value of sample is different from zero, it is because it is periodic.

3.3 Transformation rules for ROS

This section details the transformation rules for generating a code from designed applications to be executed in an ROS platform. The following figure shows templates to follow in order to generate final ROS nodes. As in previous sections, italic font is to highlight the information coming from designed applications. Although ROS permits different source code as C++, Python, and Lisp, this paper generates C++ code.

The following is detailed how develop R1…R4 rules for generating target code to be run over ROS runtime platform:

- Rule 1: Generate application dependent ROS nodes. A ROS node for each application component is required.

The declaration in the header file is

```
class RosNodeComponent[@id]  : public Component[@refAtomicCode]{
  protected:
      ros::Handle node;
  public:
      RosNodeComponent[@id] ();
      virtual RosNodeComponent[@id]();
};
```

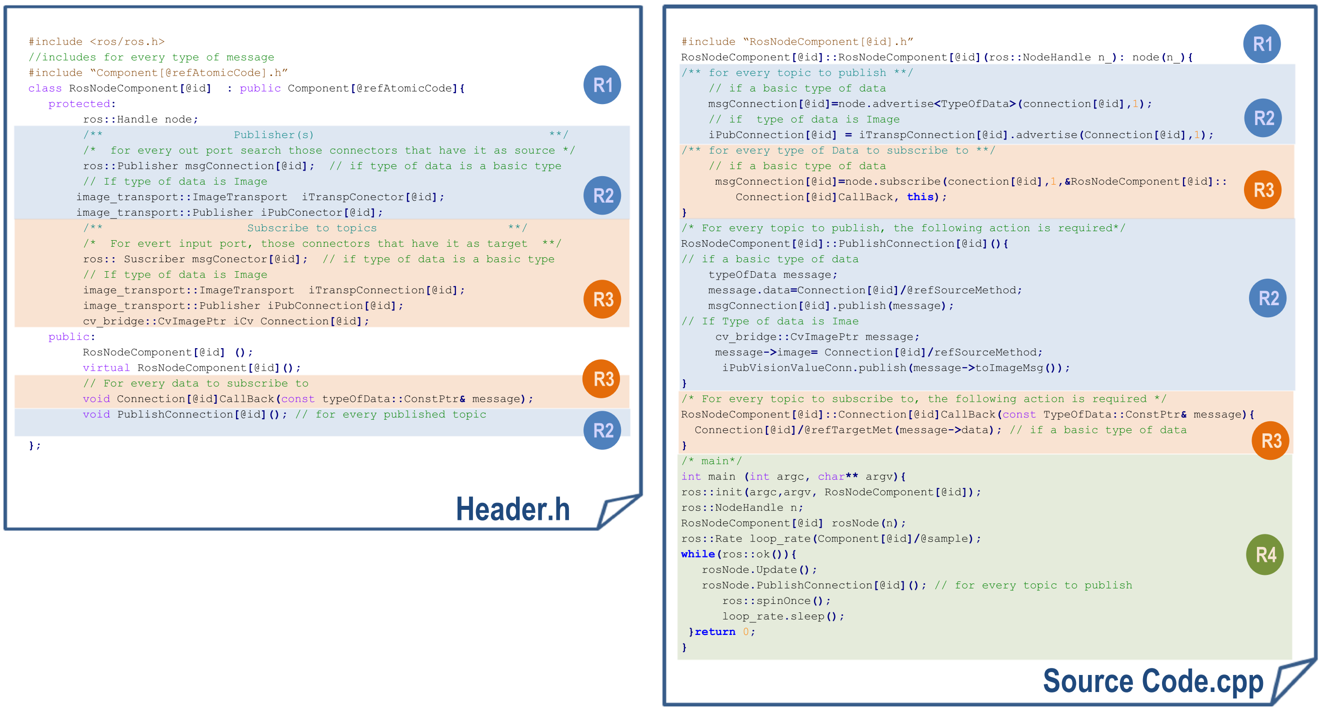
On the other hand the application-dependent node in the source file is the following class:

RosNode*Component[@id]*::RosNode*Component[@id]*(ros::NodeHandle n_): node(n_){/*code to be generated*/}.

In which Component[@id] is the name given by domain expert during the design phase to the conceptual module

- Rule 2: Publish data. Those connections in a connector which uses as source an output port of a component implies that the resulting ROS node must publish them as topics.

Hence, to publish a basic type of data, [25] implies a definition of a message in the header file. The left part of Fig. 4 with R2 details the sentences for a basic type but also for images types of data.


R1
R2
R3
R3
R2
Header.h
R1
R2
R3
R2
R3
R4
Source Code.cpp

```
#include <ros/ros.h>
//includes for every type of message
#include "Component[@refAtomicCode].h"
class RosNodeComponent[@id]  : public Component[@refAtomicCode]{
   protected:
        ros::Handle node;
        /**                   Publisher(s)                          **/
        /*  for every out port search those connectors that have it as source */
        ros::Publisher msgConnection[@id];  // if type of data is a basic type
        // If type of data is Image
       image_transport::ImageTransport  iTranspConector[@id];
       image_transport::Publisher iPubConector[@id];
        /**                     Subscribe to topics                  **/
        /*  For evert input port, those connectors that have it as target  **/
        ros:: Suscriber msgConector[@id];  // if type of data is a basic type
        // If type of data is Image
        image_transport::ImageTransport  iTranspConnection[@id];
        image_transport::Publisher iPubConnection[@id];
        cv_bridge::CvImagePtr iCv Connection[@id];
   public:
        RosNodeComponent[@id]  ();
        virtual RosNodeComponent[@id]();
        // For every data to subscribe to
        void Connection[@id]CallBack(const typeOfData::ConstPtr& message);
        void PublishConnection[@id](); // for every published topic
};
```

```
#include "RosNodeComponent[@id].h"
RosNodeComponent[@id]::RosNodeComponent[@id](ros::NodeHandle n_): node(n_){
/** for every topic to publish **/
    // if a basic type of data
    msgConnection[@id]=node.advertise<TypeOfData>(connection[@id],1);
    // if  type of data is Image
    iPubConnection[@id] = iTranspConnection[@id].advertise(Connection[@id],1);
/** for every type of Data to subscribe to **/
    // if a basic type of data
     msgConnection[@id]=node.subscribe(conection[@id],1,&RosNodeComponent[@id]::
        Connection[@id]CallBack, this);
}
/* For every topic to publish, the following action is required*/
RosNodeComponent[@id]::PublishConnection[@id](){
// if a basic type of data
    typeOfData message;
    message.data=Connection[@id]/@refSourceMethod;
    msgConnection[@id].publish(message);
// If Type of data is Imae
     cv_bridge::CvImagePtr message;
     message->image= Connection[@id]/refSourceMethod;
      iPubVisionValueConn.publish(message->toImageMsg());
}
/* For every topic to subscribe to, the following action is required */
RosNodeComponent[@id]::Connection[@id]CallBack(const TypeOfData::ConstPtr& message){
  Connection[@id]/@refTargetMet(message->data); // if a basic type of data
}
/* main*/
int main (int argc, char** argv){
ros::init(argc,argv, RosNodeComponent[@id]);
ros::NodeHandle n;
RosNodeComponent[@id] rosNode(n);
ros::Rate loop_rate(Component[@id]/@sample);
while(ros::ok()){
   rosNode.Update();
   rosNode.PublishConnection[@id](); // for every topic to publish
      ros::spinOnce();
      loop_rate.sleep();
 }return 0;
}
```

Fig. 4 Templates to follow for generating header and source code of ROS nodes

Additionally, independently of which type of data to publish, this rule finishes with a publication method for every topic to publish.

The right part of Fig. 4 with R2 indicates the different command sentences to publish a topic in the source file.

- Rule 3: Subscribe to data. Those connections in a connector which have as target an input port of a component indicate that the resulting ROS node must subscribe to such data.

The header file required sentences for declaring the data to publish by the ROS node are highlighted in the right part of Fig. 4 with R3 for a basic type but also for images type of data.

Additionally, independently of which type of data to subscribe to, a subscription method is required:

Finally, every topic to subscribe to must be initialized by the constructor. To do this, the actual parameters of the subscription method are as follows: name of the topic to subscriber to, number of data that will be stored in a buffer (1), and the method that accesses the information of the subscribed data (*Connection[@id]*CallBack). This function is responsible for the data update with the value of the subscribed topic.

The right part of Fig. 4 with R3 indicates the different code sentences to add in order to subscribe to a topic in the source file.

- Rule 4: Generate main ROS nodes. A fixed structure has been given for every ROS nodes in the source file. It follows this sequence:

– Initialization of ROS node. Definition of a node handler to communicate with the ROS Master [26].
– Definition of an instance object of the class that represents the application ROS node.

  This definition invokes a default constructor, to which the handler of node is given.

– If ROS node publishes data, it is mandatory to indicate how often the value has been updated:
– The functionality of the ROS node is performed by a loop where

  1. Execution of the atomic code encapsulated in ROS node. Publish all topics.
  2. ROS Master processes messages.
  3. ROS node is asleep till to update the value.

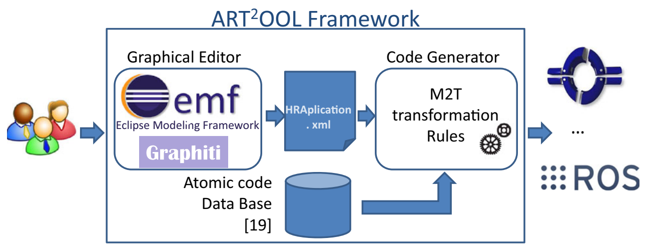


Fig. 5 General scenario of the ART2ool framework

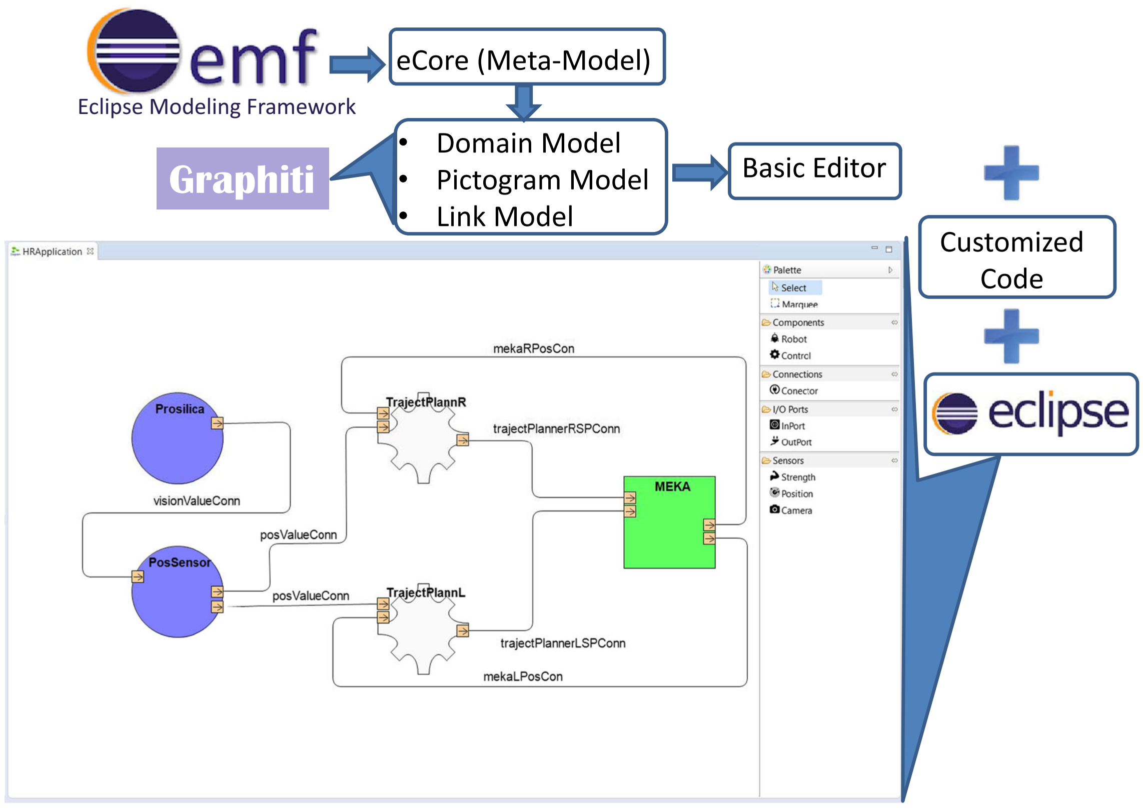


Fig. 6 Graphical modeling example

The right part of Fig. 4 with R4 lists all sentences automatically generated following this sequence.

Once generated application-dependent ROS nodes, they require to be compiled. To do this, the *CMakeList.txt* file must be composed by the following sentences for every generated ROS node:

Rosbuild_add_executable(RosNode*Component[@id]*src/RosNode*Component[@id]*.cppsrc/RosNode*Component[@id]/@refAtomicCode*.cpp);

Finally, in order to launch the application in ROS platform, a launch [26] is defined. This file is an ML that collects the package where it is, the type of node, and its name.

4 ART^{2}ool: graphical framework for generating target code automatically

Figure 5 illustrates the general scenario of the proposed framework which follows MDE principles, and it is composed mainly by two modules: (1) a graphical editor that guides the modeling of handle robot-based applications, providing an abstraction layer to the platforms where application is going to be running. (2) Code generator module that performs previous section identified rules by means of Model to Text transformation techniques of MDE. The logic of these transformation rules depends of the target runtime platform. Following sub-sections detail each module.

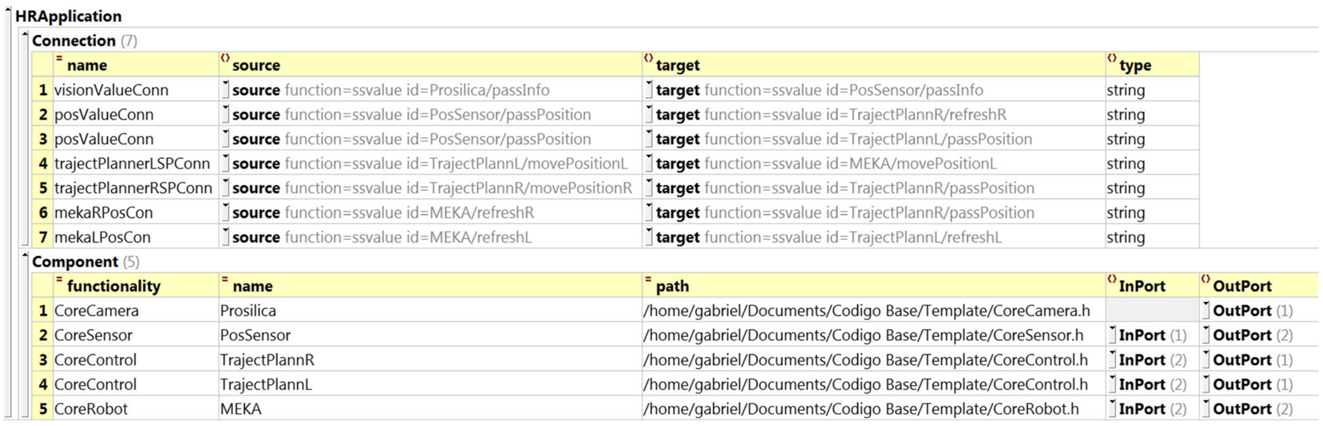
4.1 Graphical editor

The graphical editor is based on Model Driven Techniques which has been developed in Eclipse with two wide spread plug-ins: Eclipse Modeling Framework (EMF) and Graphiti [27].

On the one hand, EMF provides the basis for modeling and the facilities for automatic code generation in order to develop tools or other applications based on structured data model (ecore) stored in a ML format. To be precise, ART^{2}ool has implemented in *ecore* the meta-model shown in Fig. 2.

On the other hand, Graphiti supports a fast and easy creation of graphical editors that visualize an underlying Domain Model using a tool-defined graphical notation and is editable. This paper, concretely, has supported a Spray platform for generating Graphiti files, which are

- *Domain Model*: implements the meta-model illustrated in Fig. 2. As commented previously, this file is defined in EMF and linked to the Graphiti eclipse project. From the Graphiti point of view, this file contains the concepts which can have a graphic symbol.
- *Pictogram Model*: contains the complete information for representing a diagram. That implies that each diagram can be represented without the presence of the Domain Data. As a result, a partially redundant storage of data is required that is present both in the Pictogram Model and in the Domain Model.



HRApplication

Connection (7)

	name	source	target	type
1	visionValueConn	source function=ssvalue id=Prosilica/passInfo	target function=ssvalue id=PosSensor/passInfo	string
2	posValueConn	source function=ssvalue id=PosSensor/passPosition	target function=ssvalue id=TrajectPlannR/refreshR	string
3	posValueConn	source function=ssvalue id=PosSensor/passPosition	target function=ssvalue id=TrajectPlannL/passPosition	string
4	trajectPlannerLSPConn	source function=ssvalue id=TrajectPlannL/movePositionL	target function=ssvalue id=MEKA/movePositionL	string
5	trajectPlannerRSPConn	source function=ssvalue id=TrajectPlannR/movePositionR	target function=ssvalue id=TrajectPlannR/passPosition	string
6	mekaRPosCon	source function=ssvalue id=MEKA/refreshR	target function=ssvalue id=TrajectPlannR/passPosition	string
7	mekaLPosCon	source function=ssvalue id=MEKA/refreshL	target function=ssvalue id=TrajectPlannL/refreshL	string

Component (5)

	functionality	name	path	InPort	OutPort
1	CoreCamera	Prosilica	/home/gabriel/Documents/Codigo Base/Template/CoreCamera.h		OutPort (1)
2	CoreSensor	PosSensor	/home/gabriel/Documents/Codigo Base/Template/CoreSensor.h	InPort (1)	OutPort (2)
3	CoreControl	TrajectPlannR	/home/gabriel/Documents/Codigo Base/Template/CoreControl.h	InPort (2)	OutPort (1)
4	CoreControl	TrajectPlannL	/home/gabriel/Documents/Codigo Base/Template/CoreControl.h	InPort (2)	OutPort (1)
5	CoreRobot	MEKA	/home/gabriel/Documents/Codigo Base/Template/CoreRobot.h	InPort (2)	OutPort (2)

Fig. 7 HRApplication example

- *Link Model*: connects data from the Domain Model and the graphical representation (i.e., data from the Pictogram Model). These connections are again needed by many actions in the graphical editor. For instance, a deletion or a move of a graphical object needs also access to the associated object of the Domain Model in order to be able to make the necessary changes.

The following figure the main shows an example of an application defined with the graphical modeling editor.

Customized code includes the generation of *HRApplication.xml* file which is the output of this module and the input for the code generator. Figure 7 the *HRApplication* XML file graphically designed in Fig. 6.

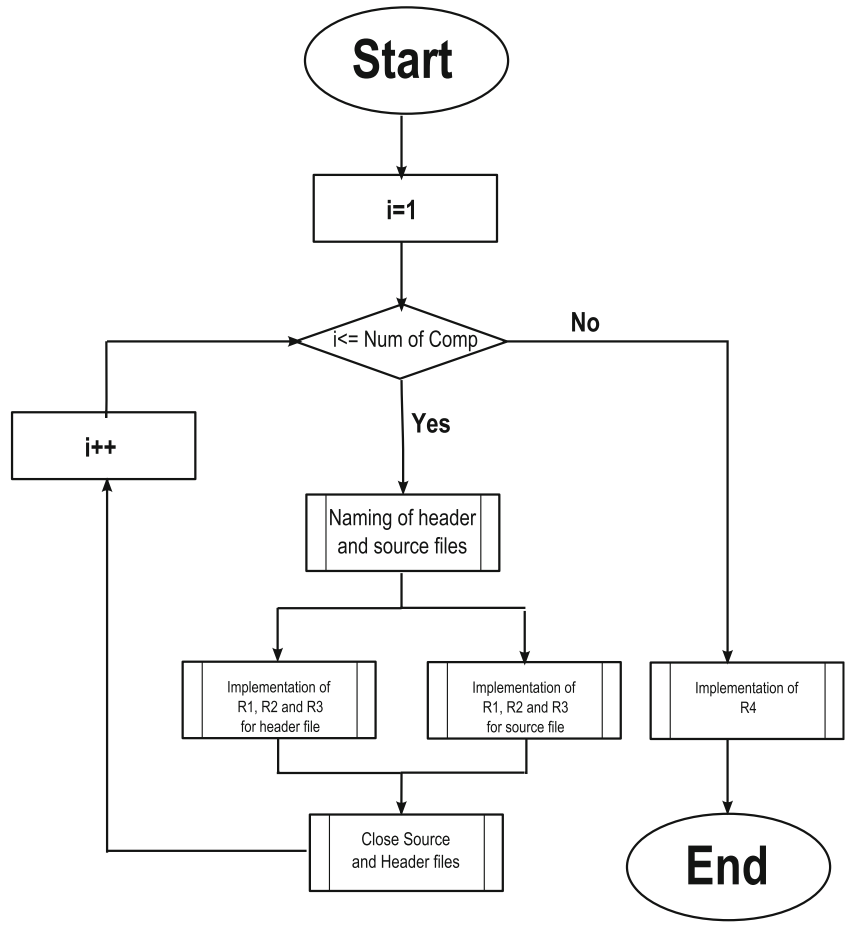

Start
i=1
i<= Num of Comp
No
Yes
i++
Naming of header and source files
Implementation of R1, R2 and R3 for header file
Implementation of R1, R2 and R3 for source file
Implementation of R4
Close Source and Header files
End

Fig. 8 General flow followed by code generators

Fig. 9 Main function for generating OROCOS application-dependent components

```
public void genOrocosFiles(){
    String orocos = "Orocos";
    String fileName;
    String atomicClassName;
    Element component;
    ArrayList<String> typeOfProperties;
    // instance of the Class that implements methods to generate Deployment XML
    DeploymentFileGenerator deployment = new DeploymentFileGenerator();
    for (int i = 0; i < getcomponents().getLength(); i++) {
        component = (Element)getcomponents().item(i);
        fileName = orocos.concat(component.getAttribute("name"));
        atomicClassName = component.getAttribute("refAtomicCode");
        // header and source files naming
        initOrocosFiles(fileName);
        typeOfProperties = getTypeOfProperties(component.getAttribute("name"));
        /* Header file generation */
        genIncludesFiles(atomicClassName, typeOfProperties);
        // Implemetation of Rule 1
        genClassDeclaration(fileName, atomicClassName);
        //Implementation of Rule 2
        genOutputPortVars(getOutputPortNames(component), typeOfProperties);
        //Implementation of Rule 3
        genInputPortVars(getInputPortNames(component), typeOfProperties);
        genOrocosDeclaration(fileName);
        /* Source file generation */
        genClassConstructor(fileName, atomicClassName); // Rule 1
        genLabelForPublishPorts(fileName, getOutputPortNames(component)); //Rule 1
        //Escribe las funciones en el fichero .cpp
        genOrocosMethods(fileName); //configureHook, start, stop and header of updateHook
        //Implementation of Rule 2
        genPublishData(getOutputPortNames(component));
        //Implementation of Rule 3
        genSubscribeData(getInputPortNames(component));
        genOrocosPrimitives(fileName); // Rule 1
        closeOrocosFiles();
    }
    deployment.generate(); // generation of Deployment XML File
}
```

4.2 Code generator

Before starting with code generation, it is necessary indicate the target runtime platform. This module processes previously generated *HRApplication.xml* file (see Fig. 5) with a set of M2T transformation rules. It is based on Java and DOM (Document Object Model) for managing the input information (HRAplication model) (Fig. 7). As stated previously, the current version supports generation by OROCOS and ROS. The flow followed by these two generators is the following (Fig. 8):

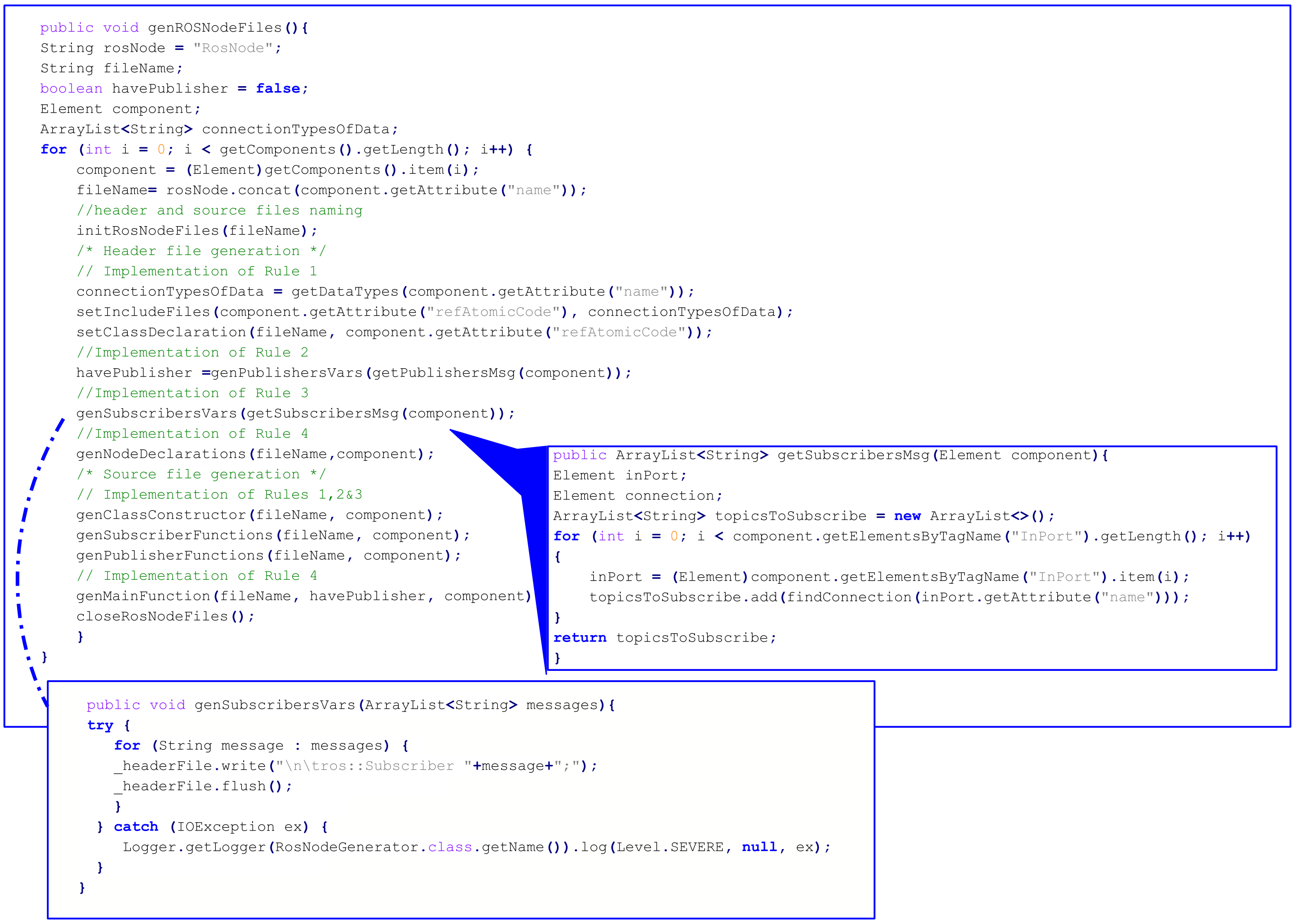
Figure 9 shows the general algorithm for generating OROCOS application components (*genOrocosFiles*). For instance, to generate the header file, in addition for naming issues, it is also necessary to define *genOutputPortVars* and *genInputPortVars* functions, which perform rules R2 and R3, respectively (i.e., they define the ports of the OROCOS application component). Source file generation involves seven functions, five for performing R1, one for R2, and the other for performing R3.

On the other hand, Fig. 10 shows the main logic to generate target code for the ROS platform (*genROSNode*). The algorithm starts processing all components of the logic. For every application component, a RosNode is generated with two files (.h and .cpp).

For instance, to generate a header, a set of functions has been developed to implement every transformation rule. Figure 10 details the *genSubscribersMsg* function which returns a list of topics to subscribe to. In fact, it processes every *InPort* of the component and adds its name as a topic to subscribe to.

5 Case study: tracking an object in movement

Object tracking is a quotidian task that humans perform easily, but when this task is performed by a robot, it is not trivial for many reasons. On the one hand, it is necessary to have a complete knowledge of the environment where the robot can move. It is important to emphasize that sometimes there can be object-obstruction problems. On the other hand, once the object is located, a trajectory control must be defined that allows tracking the object successfully, avoiding collision with other objects present at the scene.



```
public void genROSNodeFiles(){
String rosNode = "RosNode";
String fileName;
boolean havePublisher = false;
Element component;
ArrayList<String> connectionTypesOfData;
for (int i = 0; i < getComponents().getLength(); i++) {
    component = (Element)getComponents().item(i);
    fileName= rosNode.concat(component.getAttribute("name"));
    //header and source files naming
    initRosNodeFiles(fileName);
    /* Header file generation */
    // Implementation of Rule 1
    connectionTypesOfData = getDataTypes(component.getAttribute("name"));
    setIncludeFiles(component.getAttribute("refAtomicCode"), connectionTypesOfData);
    setClassDeclaration(fileName, component.getAttribute("refAtomicCode"));
    //Implementation of Rule 2
    havePublisher =genPublishersVars(getPublishersMsg(component));
    //Implementation of Rule 3
    genSubscribersVars(getSubscribersMsg(component));
    //Implementation of Rule 4
    genNodeDeclarations(fileName,component);
    /* Source file generation */
    // Implementation of Rules 1,2&3
    genClassConstructor(fileName, component);
    genSubscriberFunctions(fileName, component);
    genPublisherFunctions(fileName, component);
    // Implementation of Rule 4
    genMainFunction(fileName, havePublisher, component);
    closeRosNodeFiles();
    }
}
```

```
public ArrayList<String> getSubscribersMsg(Element component){
Element inPort;
Element connection;
ArrayList<String> topicsToSubscribe = new ArrayList<>();
for (int i = 0; i < component.getElementsByTagName("InPort").getLength(); i++)
{
    inPort = (Element)component.getElementsByTagName("InPort").item(i);
    topicsToSubscribe.add(findConnection(inPort.getAttribute("name")));
}
return topicsToSubscribe;
}
```

```
public void genSubscribersVars(ArrayList<String> messages){
  try {
     for (String message : messages) {
     _headerFile.write("\n\tros::Subscriber "+message+";");
     _headerFile.flush();
     }
  } catch (IOException ex) {
     Logger.getLogger(RosNodeGenerator.class.getName()).log(Level.SEVERE, null, ex);
  }
}
```

Fig. 10 Main function for generating ROS application-dependent nodes

This case study describes how the Meka humanoid robot performs a tracking task of an object in movement with the two arms. This implies not only a 3D object location but also two-arm trajectory control to avoid any type of collision with a conveyor belt or between themselves.

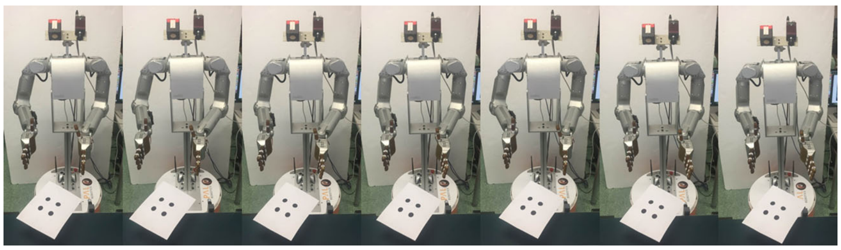
To perform this tracking task, Prosilica GX1050 has been used as a vision sensor that has detected the object's 3D position jointly with other processing algorithms. In order to simplify the case study, this task will be based on a partially known environment, because the position of obstacles such as the conveyor belt is known. Hence, this task starts with the identification of the object to track. Then, a Meka robot adjusts the position of its arms depending on the 3D position provided by the computer vision system.

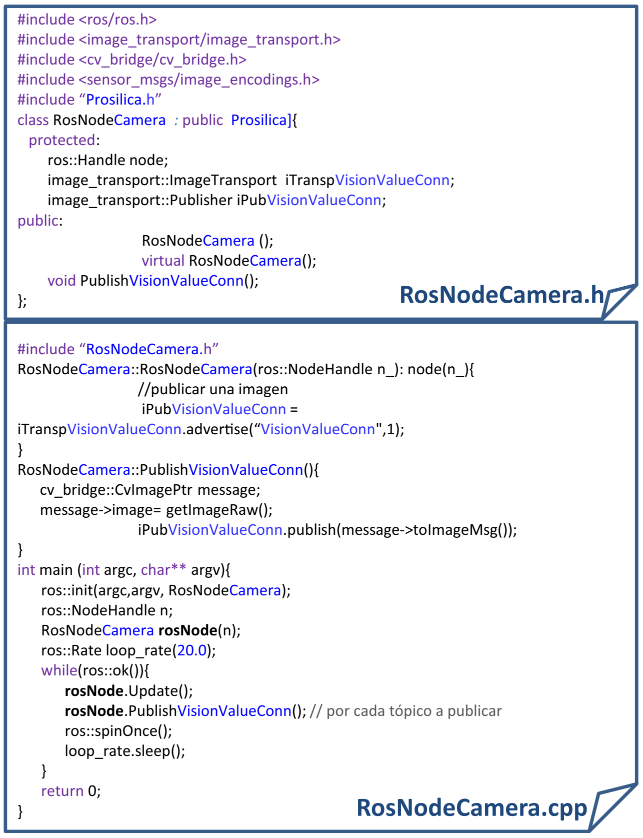
Figure 11 shows a sequence of movements of the Meka robot. As demonstrated previously, the object has been labeled with a known pattern and with a conventional camera jointly with four processing points: a processing algorithm allows locating the object.

Figure 6 shows the functionality of the application modeled with the ART^{2}ool. The target platform for this example is ROS.

Figure 12 details target code generated automatically to the definition of camera ROS node. As can be seen, this node publishes 20 images per second, because the time spent between two images acquisition and processing is 50 ms, the camera takes 30 ms in the acquisition and the processing algorithm 20 ms.

Fig. 11 Tracking of an object in movement by the Meka robot





```
#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

Fig. 12 Target ROS code for the RosNode camera

In order to launch the application, the launch file with the list of nodes to start running is also automatically generated.

Finally, the authors want to note that the atomic code encapsulated in application components is reused in other applications that run over OROCOS [28]. More specifically, the four-point processing algorithm that provides a position, having as input an image, and a trajectory control algorithm have been reused in two very different handle robot-based applications.

6 Conclusions

A platform that provides support for the development cycle of handle robot-based applications has been proposed, following MDE principles. In particular, guidelines for designing these applications have been provided, guaranteeing the reuse of the atomic code for applications, independently of the platform they will be running. The ART^{2}ool is very useful for application domain experts, because it guides them along the design of the logic, abstracting them from the emerging techniques.

On the other hand, guidelines of MDE have been followed for the automatic code generation M2T. In summary, the current version of ART2tool provides support for component-based robotics middleware and for ROS.

Funding information This work was financed in part by the MCYT&FEDER under DPI2016-78290-R.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

1. European Commission: Research and Innovation (2013) Factories of the future PPP: towards competitive EU manufacturing. European Union
2. BlanchetM, RinnT, Von ThadenG, ThieulloyG (2014) Industry 4.0: the new industrial revolution—how Europe will succeed
3. National Science and Technology Council (2016) Advanced manufacturing: a snapshot of priority technology areas across the Federal Government Subcommittee for Advanced Manufacturing
4. RobMoSys (2017) Composable models and software for robotic systems [online]https://robmosys.eu/about/
5. Selic B (2003) The pragmatics of model driven development. IEEE Softw 20(5):19–25
6. Brugali D, Scandurra P (2009) Component-based robotic engineering (part I) reusable building blocks. IEEE Robotics Automation Magazine 16(4):84–96
7. Brugali D, Shakhimardanov A (2010) Component-based robotic engineering (part II) systems and models. IEEE Robotics Automation Magazine 17(1):100–112
8. BarnerS., GeisingerM., BucklC., KnollA. (2008) EasyLab: model-based development of software for mechatronic systems. Proc IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications, pp:540–545
9. 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, pp: 135–146
10. Commission IEC (2013) International Standard IEC 61131-3, Programmable Logic Controllers Part 3
11. AlonsoD., Vicente-ChicoteC., OrtizF., PastorJ., ÁlvarezB. (2010) V3CMM: a 3-view component metamodel for model-driven robotic software development. J Software Eng Robot, pp: 3–17
12. 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, pp: 324–335
13. GarciaH, BruyninckxH (2014) Tool Chain (BRIDE) delivered as BRICS software distribution. [online]http://www.best-of-robotics.org/bride/
14. Steinberg D, Budinsky F, Paternostro M, Merks E (2008) EMF: eclipse modeling framework, 2nd ed. Addison-Wesley Professional
15. Bruyninckx H. (2001) Open robot control software: the OROCOS. Proc. of IEEE Int. Conf. on Robotics and Automation (ICRA), pp: 2523–2528
16. Jesse Russell, Ronald Cohn (2012) ROS (robotic operating system, VSD
17. Kumar PS, Emfinger W, Kulkarni A, Karsai G, Watkins D, Gasser B, Ridgewell C, Anilkumar A (2015) ROSMOD: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ROS. Proc. of Rapid System Prototyping Symposium, pp 1–7
18. Reusable Software Apps for Robotic Applications (ReApp) (2018), [website] http://www.reapp-projekt.de/index.php?id=reapp_project
19. 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

20. Álvarez B, Ortiz F, Pastor JA, Sánchez P, Losilla F, Ortega N (2006) Arquitectura para control de robots de servicio teleoperados. Revista Iberoamericana de Automática e Informática Industrial 3(2):79–89
21. Garcia GJ, Corrales JA, Pomares J, Torres F (2009) Survey of visual and force/tactile control of robots for physical interaction in Spain. Sensors 9:9689–9733
22. DeliverableD-2.1 Best practice assessment of software technologies for robotics, [Online] Available: http://www.best-of-robotics.org/pages/publications/BRICS_Deliverable_D2.1.pdf
23. OpenRTM. [Online] Website: http://www.openrtm.org/openrtm/en/node/780
24. 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
25. ROS msg. [Online] http://wiki.ros.org/msg
26. Martínez A, Fernández E (2013) Learning ROS for robotics programming. Publishing ltd., Packt
27. Graphiti (2016) [online] https://eclipse.org/graphiti/documentation/
28. Estévez E, Sánchez-García A, Gámez-García J, Gómez-Ortega J, Satorres- Martínez S (2016) A novel model-driven approach to support development cycle of robotic systems. Int J Adv Manuf Technol 82(1):737–751