

A novel model-driven approach to support development cycle of robotic systems

Elisabet Estévez¹ · Alejandro Sánchez-García¹ · Javier Gámez-García¹ · Juan Gómez-Ortega¹ · Silvia Satorres-Martínez¹

Received: 30 December 2014 / Accepted: 5 June 2015 / Published online: 23 June 2015
© Springer-Verlag London 2015

Abstract Currently, industrial robots are decisive in modern production facilities, and in a near future, robots will also become essential in daily life. In fact, the main aim of robotic manipulator relies on the integration of robots into people's daily. To this purpose, there are a great number of physical devices, such as sensors, actuators, auxiliary elements, tools etc. which can be incorporated into a robot. Although integration, reuse, flexibility and adaptability are crucial characteristics demanded by current robotic applications, there is a lack of standardization in terms of hardware and software platforms, providing incompatible task-specific and non-reusable solutions. Consequently, there is a need for a new engineering methodology to design, implement and execute software systems. This work explores the advantages that model-driven engineering provides for the development of applications for robotic manipulators' platforms. Specifically, a modelling approach is developed to generate the target code automatically. To validate the proposal, a tool that allows the final code to be generated for most spread communication middlewares in the robotics field is also presented.

Keywords Robotic arm manipulators · Model-driven engineering · ROS—robotic operating system · OROCOS—open robot control software

1 Introduction

In the never-ending effort of humanity to simplify their existence, the introduction of *intelligent* resources was inevitably going to come up. One characteristic of these intelligent systems is their ability to adapt themselves to the variations in the outside environment as well as to the internal changes occurring within the system. In other words, these systems can be considered dynamical systems. In this sense, robotics discipline is closely related to the construction of such intelligent systems because it has as its aim the construction of robots that can work in an automatic way performing even difficult tasks for humans. Robotics discipline is decisive in modern production facilities and, in the near future, in daily life tasks. There are many different kinds of robots available, each one created for different tasks and behaviours, and to work on different platforms. Robots can be built for entertainment, knowledge, competitions, household chores, industrial uses etc. This work is focused on robotic manipulator-based applications where every robotic platform, besides the robot, also requires a set of sensors to take information from the environment together with controllers. [1, 2] are two application examples with very different hardware and software solutions, required when a manipulator arm has to interact with its environment. Authors in [3] present a platform, based on contactless technology, where a flexible robotic manipulator identifies and characterizes different work pieces travelling on a conveyor. In [4], a complete concept and design of a novel friction stir welding robotic platform for welding polymeric materials is presented. [5] defines a reconfigurable robot multi-axis machining system for machining complex ports of light materials with lower tolerances, with freeform surfaces.

Regardless of the task, every robotic arm application needs to get information, through sensors, from the varying and unknown environment; later on, this information is used by

✉ Elisabet Estévez
eestevez@ujaen.es

¹ Departamento de Ingeniería de Electrónica Automática, Universidad de Jaén, Jaén, Spain

controller(s) or algorithm(s), which also include data about the robot state (position, velocity, acceleration, torque ...).

Although many elements, such as sensors, actuators, auxiliary elements (e.g. processing algorithms), tools etc., need to be added to a robot to make it more flexible and adaptable, their integration and collaboration is not easy to manage. Normally, the software development methodology followed lacks crucial characteristics such as reusability, i.e. the ease with which a system can be modified to be used in applications or environments that are different from those which it was originally designed for. This is much related to flexibility, adaptability and standardization in terms of hardware and software platforms [6]. According to this, new robotic software infrastructures should allow developers to face up to the complexity which is imposed by different issues such as hardware, software, real time and distributed computing environments. Currently, two major problems can be found. The first one is due to the ad hoc solutions and closed products provided by manufacturers. The second relies on the lack of standardization in terms of hardware and software platforms to achieve the features demanded by applications. For the latter, the fashion using of ICT Information and Communication Technologies of Software Engineering is starting to be adopted in the robotics field. In fact, as in other fields, a strong movement toward software engineering principles is also taking place in robotics [7, 8]. For example, authors in [9] propose the use of event-driven function blocks for robotics assembly tasks in real time in order to improve the adaptability and flexibility of robotic assembly systems. The use of the component-based software engineering (CBSE) is not new in the robotics field [10, 11]. The CBSE discipline is used in order to tackle the complexity due to the variability of hardware devices and software components in robot-based applications. As is well known in the software engineering community, the CBSE offers mechanisms for increasing the abstraction level, so applications can be developed as a set of independent modules which interchange information with each other by their interfaces. [12–15] are examples of work which present CBSE-based approaches offering a high rate of reusability. The degree of reusability could be high once the communication middleware is selected, i.e. the code encapsulated in such components is middleware dependent. Thus, CBSE-based approaches offer little flexibility for assuring reusability even regardless of the communication middleware.

The model-driven paradigm increases the abstraction layer, allowing the description of applications completely regardless of the software platform [16]. The key concept of this paradigm is the model, where the main concepts of the systems are

detailed. Models are used in all phases of the system development cycle, until the generation of the target code [17, 18]. Thus, models detail the information of an application from a specific domain point of view but they also act as input and output of every phase for the development cycle of the system [19].

The use of such techniques is also being introduced in robotics [14]. For example, [20] gives a demonstration for the application of verification by model checking to substantial control intensive application developed in a commercially supported and widely used object-oriented (OO) development process. [21] proposes a modelling language to formally model the solution space and to specify the quality attributes during design time. Hence, this is a complementary approach that helps in early analysis of quality attributes, to identify variations and act as a bridge between problem and implementation space. [22] proposes two model-based programming tool that generates a target source code for Robotino® mobile platform. This tool allows users to define the functionality of the task (application logic model) and the required hardware resources for the specific platform of Robotino. Authors in [23] present *V3CMM—3 View Component Meta-Model* in order to generate ADA skeleton code to control a Cartesian robot. More recently, [24, 25] detail SmartSoft MDSD—model-driven software development—toolchain for modelling robotic applications with SmartSoft component concept [26, 27]. The generated target code runs specifically over CORBA middleware [28], so this target code cannot run over any other communication middleware.

Additionally, [29, 30] describe the main goal of BRICS—best practice in a robotics—European project that consists of structuring and formalizing the robot development process. BRIDE (BRICS Integrated Development Environment) allows defining the robotics applications to be defined using OROCOS (open robot control software) components [31] or ROS (robot operating system) nodes [32]. This is a very interesting modelling approach but in this case also valid only for ROS and OROCOS.

In order to achieve total reusability, this work goes one step further. It is inspired by model-driven architecture (MDA) standard [33], so firstly, the functionality of the application is defined, i.e. a platform-independent model (PIM) that contains *what* the system has to do independently of how and where it will be running; afterwards, the hardware and software platform features are fixed in the platform-specific model (PSM); finally, the target code could be automatically generated. Hence, the same functionality of a robotic application can be executed in different platforms. The model-driven engineering (MDE) discipline offers model to model (M2M) transformation techniques as very powerful mechanisms for generating the target PSM from the PIM. Hence, different PSMs can be generated from a unique PIM. Furthermore, model to text transformation techniques allow generating the

skeleton code to be generated automatically. In consequence, the use of those mechanisms, offered by MDE, guarantees the achievement of complete reusability, so the support of application's variability and interoperability is assured.

Moreover, the proposed approach, which is based on those mechanisms, makes use of the UML (Unified Modelling Language) [34] standard notation to describe the system's functionality as well as its platform. [35, 36] propose a UML-based robotics architecture for LEGO and Claw Car Robots respectively. [37] uses UML notation for specifying the requirements, documenting the structure and defining the relationships between objects especially in service robot systems. This work extends to a previously mentioned work because the information is also available in XMI (XML Metadata Interchange) [38] standard format model which acts as input model to M2M and model to text (M2T) transformation techniques, applied in order to generate the final source code.

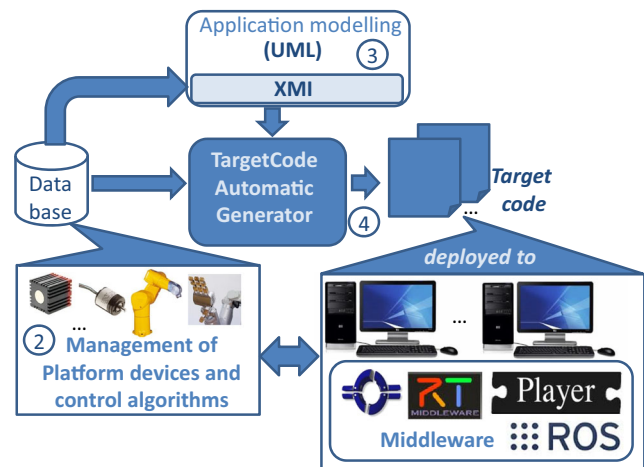
In consequence, the main contributions of this paper are the following: (1) a UML-based modelling approach for designing robotic arm applications. This modelling provides support to the design phase of the development cycle of such applications. The result of the design phase is a XMI model with functionality and hardware/software (HW/SW) information. (2) Identification and development of M2T transformation rules to generate target source code. The M2T transformations allow the same functionality to be run over the most widespread communication middlewares in the robotics community. Thus, a complete development support for the robotic system is achieved.

The remainder of this work is as follows: Section 2 describes a UML-based methodology for modelling robotic arm applications. Section 3 presents the main features of most widespread communication middleware in the robotics field. This section concludes with the main rules for achieving the automatic code generation. The proposed modelling approach has been tested in Section 4 with two very different case studies, one for an industrial robot and the other for service robot. Finally, Section 5 introduces the conclusions of this work.

2 Framework for modelling robotic arm applications

The general scenario of the proposed framework to provide support of the development cycle for robotic arm industrial production applications is illustrated in Fig. 1.

The first phase relies on the specification of functional and non-functional *requirements*. The analysis phase (Fig. 1) is responsible for selecting the application control strategies. The codification of the device management in an isolated form (i.e. without taking into account the logic/behaviour of applications) and software algorithms must also be done during this phase. The latter are stored in a database.



② Analysis Phase ③ Design Phase ④ Code Generation
Fig. 1 General scenario of the model-based proposed framework

Previous work of the authors analyses the main components involved in every robotic arm application (e.g. sensors for getting information from an unknown environment, manipulators and algorithms such as trajectory planners). A common interface to manage each kind of component is detailed in [39]. These interfaces provide designers with an abstraction of manufacturer driver knowledge; thus, they give designers common methods for managing the element in an application. Figure 2 shows the common interface for image sensors (*cameras*), where manufacturer-specific information is highlighted in purple and common methods are highlighted in green.

As mentioned above, the UML standard modelling language has been used during the design phase. Because on modelling proposed approach follows MDA guidelines, firstly, the behaviour of the application is defined (PIM), and secondly, the implementation of this behaviour, in terms of software and hardware platforms, are detailed. Hence, the modelling approach consists of two main steps: (1) the definition of the application functionality and (2) the characterization of the platform where it will be deployed and is going to be running.

The following sub-sections detail the steps to follow in order to provide support to the design phase in the proposed framework (see Fig. 1).

2.1 Import of required atomic code interfaces to UML

This sub-section is required in order to supply the interfaces of every atomic code, stored in a repository, to a UML modelling tool. To do this, XMI standard notation has been used [40]. Object Management Group presented XMI standard to give inter-changeability among UML tools. Thus, XMI files collect information about all UML elements as well as printing information of diagrams in a markup language (ML) notation. In consequence, the authors have selected this format to import

Fig. 2 Interfaces of Guppy F-080C and Prosilica GX1050

```

class Camera {
protected:
    int w,h,shutter, gain;
    std::vector<char> imageRaw;
    std::vector<char> imageGray;
    std::vector<char> imageColor;
public:
    Camera();
    virtual ~Camera();
    virtual Status configure();
    virtual Status start();
    virtual Status capture();
    ...
};

class Guppy80 :public Camera{
private:
    dcl394camera_t *handle;
    dcl394featureset_t parameter;
    dcl394video_frame_t *frame;
    void capture();
public:
    Guppy80();
    virtual ~Guppy80();
    Status setWidth(int value);
    Status setHeight(int value);
    Status setShutter(int value);
    Status setGain(int value);
    Status configure();
    Status start();
    Status stop();
};

class GX1050 :public Camera {
private:
    tPvHandle *handle;
    tPvFrame *frame;
    void capture();
public:
    std::string ip;
    GX1050();
    virtual ~GX1050();
    Status setWidth(int value);
    Status setHeight(int value);
    Status setShutter(int value);
    Status setGain(int value);
    Status configure();
    Status start();
    Status stop();
};
    
```



templates into the UML project before the start of the design of a robotic arm application. In particular Altova Umodel UML 2.x compliant tools have been used. As UML is based on an object-oriented (OO) paradigm, the meaning of UML class is the same as any high level OO programming language (e.g. C++, java). Table 1 illustrates how the main concepts of the OO paradigm are expressed in XMI. The proposed approach gets the information, is stored in a database and generates the required XMI file to be imported by the UML modelling tool.

The structure of the generated XMI is the following: each UML lexicon is expressed in XMI as packagedElement. The xmi:type attribute's value gives the semantics to the XMI

concept. For instance, class is a packagedElement with xmi:type="uml:Class". Additionally, identifier (xmi:id), name and isAbstract attributes characterize the class. The OO properties are expressed as ownedAttribute. Every property is characterized by its name, identifier and visibility (*public*, *protected* or *private*). OO operations are expressed as ownedOperations. The inheritance is expressed with a generalization element. The general attribute collects the identifier of class the information is inherited from.

Figure 3a illustrates the XMI file highlighting the features of Guppy F-080C code interface. Figure 3b shows the same information once the XMI file is imported to UML tool.

Table 1 Mapping between OO and XMI markup language notation

OO	UML notation	XMI notation
Class		<code><packagedElement xmi:type="uml:Class" xmi:id="id" name="ClassName" isAbstract="true"> </packagedElement></code>
Property		<code><ownedAttribute xmi:type="uml:Property" xmi:id="id" name="PropertyName" visibility="public protected private"/></code>
Operation		<code><ownedOperation xmi:type="uml:Operation" xmi:id="id" name="MethodName" visibility="public protected private"/></code>
Parameter		<code><ownedParameter xmi:type="uml:Parameter" xmi:id="id" name="paramName"/></code>
Inheritance		<code><generalization xmi:type="uml:Generalization" xmi:id="id" general="id_Class1"/></code>

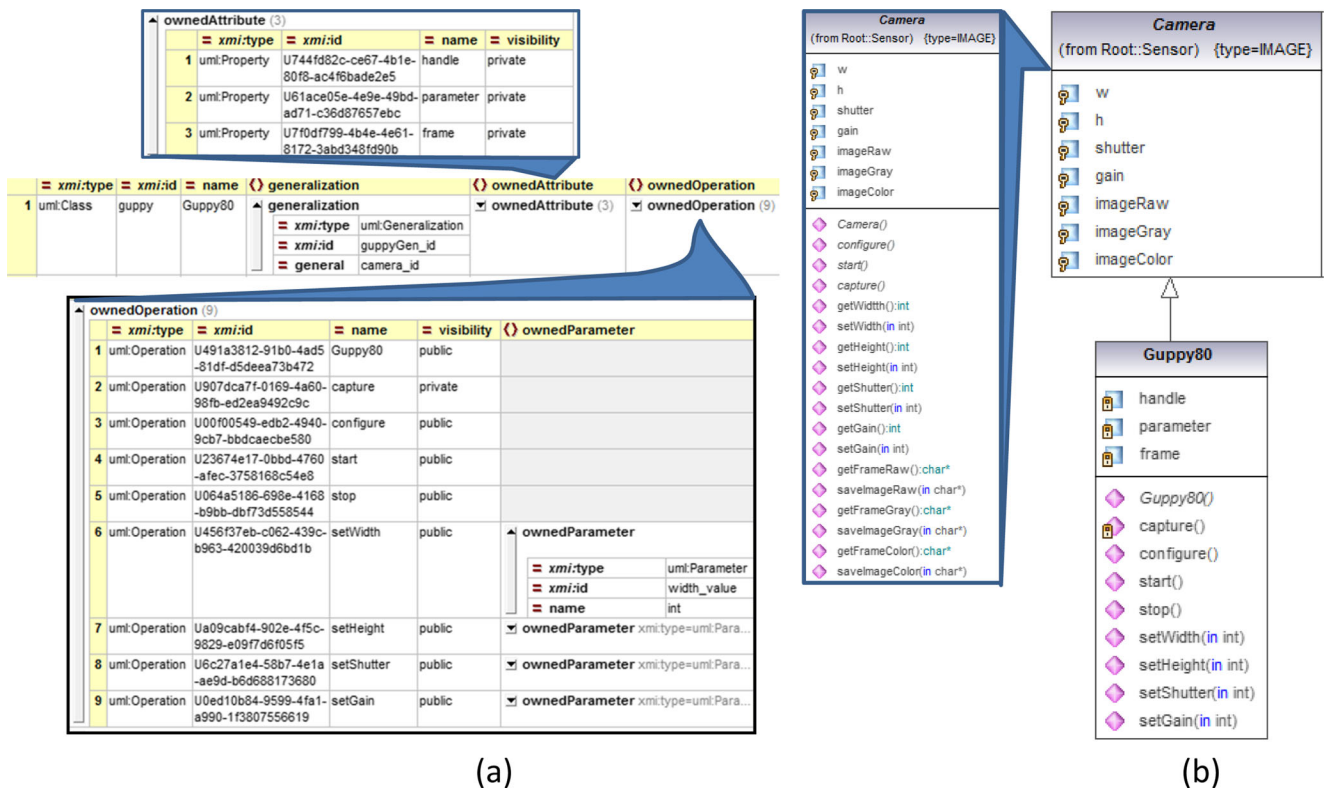


Fig. 3 Guppy F0808-C management interface in XMI (a) and UML (b) formats

2.2 Definition of application behaviour

This section defines the PIM for robotic arm applications. This behaviour has been specified with a UML 2.x component diagram. Each UML component, at the end of the design phase, will represent software component(s). Actually, the encapsulated code is expressed with a UML class, imported from the database (e.g. Fig. 3b). The *ComponentRealization* UML concept is used for indicating the code that a UML component encapsulates.

The communication (data flow) among components is achieved with UML port and interface concepts. The former provides the external accessibility point, and the latter indicates the accessible information as well as the accessibility permission (read, read-write). The accessibility of information is achieved with UML *InterfaceRealization*. The write-only external access permission is attained with UML *usage* concept. A UML port uses a UML interface to update the property’s value. Table 2 summarizes the UML elements that comprise the modelling of the functionality of robotic arm applications.

Table 2 UML elements for modelling application logic

UML concept	Graphical notation	Role in robotic arm application modelling
Component		Robotic application component
Port		Provide external accessibility to protected properties
Interface		
Provided (<i>InterfaceRealization</i>)		Read- only external access
Required (<i>Usage</i>)		Write-only external access

2.3 Platform-specific feature modelling

This section details the main UML elements used for the definition of the hardware platform where the selection of the communication middleware is also included. A communication middleware (MW) is responsible for guaranteeing communication among software components and applications independently of where they are deployed. Thus, MW could be considered a layer that lies between application code and runtime infrastructure.

Table 3 illustrates the UML elements that comprise the modelling of the deployment platform. The UML deployment diagram is formed by as many UML *device* elements as devices in the real platform. Additionally, at least one *node*, where to deploy the software application, is required. The communication protocols between devices and nodes are expressed with *CommunicationPath* UML elements. The *artefact* UML element is used to indicate the necessity of other codes or libraries in a node; so in this work, the user (modeller) should indicate information about the repository's path that collects atomic source code (see Fig. 1). Finally, the *ExecutionEnvironment* UML element is used to provide information about the communication middleware.

3 Automatic generation of target code for robotic arm applications

This section is centred on 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 input

model to the transformer is an XMI file obtained as a result of the design phase. Eventually, the generated target code should be compiled and deployed into the corresponding platform. Therefore, the definition of M2T transformation rules implies having knowledge about the structure of XMI model as well as about the particularities of selected MW.

The following sub-sections detail the main characteristics of the XMI model and main features of the most widespread communication middlewares in the robotics field. Finally, this section concludes identifying the transformation rules which are implemented by the proposed approach.

3.1 Structure of the input model

The input model, from which the code generator gets information to automatically generate the target code, is obtained exporting the resulting UML model to XMI standard notation.

Table 4 summarizes how the elements used for modelling the functionality and hardware platform of the application are expressed in the ML according to XMI meta-model.

Every application is formed by a set of interconnected components which provide logic to the application. The encapsulated code is indicated in realizingClassifier attribute of the realization element. Those components are interconnected via *provided* and *required* interfaces related with their output and input ports, respectively. Port components are detailed in XMI file as ownedAttributes. In order to know if a port is an input or an output data port, the *usage* UML element is checked.

Regarding platform-specific information, hardware platform is defined by a set of UML nodes and devices. The communication protocol between those elements implies the use of *communicationPath* UML element.

Table 3 UML elements for modelling platform-specific model




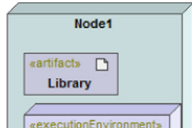
UML concept	Graphical notation	Role in robotic arm application modelling
Device		Device of robotic arm platform
Node		Node (e.g. PC) of robotic arm platform
CommunicationPath		Communication protocol between nodes, devices and node-device(s)
Artefact		Library with device's isolated management code
Execution environment		Communication middleware

Table 4 XMI notation to be processed by code generator

Application functionality (PIM)	Platform specific (PSM)
<p>Functional module</p> <pre> 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 </pre>	<p>Node</p> <pre> 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 </pre>
<p>Provided information</p> <pre> packagedElement ├── xmi:type uml:Interface ├── xmi:id id ├── name interface_Name └── ownedOperation xmi:type=uml:Operation xmi:id=id name=op_Name visibility=public </pre>	<p>Device</p> <pre> packagedElement ├── xmi:type uml:Device ├── xmi:id Device_id ├── name Device_name </pre>
<p>Required information</p> <pre> packagedElement ├── xmi:type uml:Usage ├── xmi:id id ├── supplier xmi:idref=interface_id_ref └── client xmi:idref=port_id_ref </pre>	<p>Communication bus</p> <pre> packagedElement ├── xmi:type 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 </pre>
	<p>Libraries</p> <pre> packagedElement ├── xmi:type uml:Artifact ├── xmi:id Artifact_id ├── name RequiredLibrary </pre>
	<p>MW</p> <pre> packagedElement ├── xmi:type uml:ExecutionEnvironment ├── xmi:id EE_id ├── name CommunicationMiddleware </pre>

Finally, artefact and execution environment UML elements are used for detailing required libraries and selected MW, respectively.

3.2 Main features of selected communication middleware

The M2T generator module also needs knowledge about the target code structure and requirements. Section 2.1 of [41] summarizes the main characteristics of the most widespread robotics-specific communication middleware platforms:

- OROCOS general purpose modular framework for robot and machine control [31]. This framework provides a set of libraries from which the real-time toolkit (RTT) is the main one, providing the infrastructure and functionality to build component-based real-time (RT) applications.
- *OpenRTM* [42] is an open-source implementation of the RT middleware specification and it is developed by AIST, Japan. It provides three main specifications from which the RT component framework is the principal that offers a set of classes, which can be used to develop stand-alone software components.

- Player is a software package developed at the University of Southern California [43]. It can be viewed as an application server interfacing with robot hardware devices and user-developed client programmes.
- ROS [32] is a set of open-source software libraries and tools that help to build robot applications. This began at Stanford University and Willow Garage in 2007, which has become one of the de facto standards of robotics platforms. ROS-INDUSTRIAL [44] is based on ROS but specifically for industrial robotics platforms.

The first three examples are component-based platforms. Table 5 summarizes these concepts in terms of the interface of components as well as their timing information. Every one offers an interface for achieving synchronous and asynchronous operations. Furthermore, OROCOS and OpenRTM offer *DataPorts* in order to provide feasibility to data interchanging while player offers *DataInterfaces*. Finally, those features modifiable in runtime are defined as *properties* in OROCOS, *configuration interface* in OpenRTM and *service port* in player platform.

Although ROS is not a component-based platform, it is also formed by modular programming units which are called nodes. In order to give external accessibility, the *topic* concept

Table 5 Main properties of component communication MW

Concept	Component-based middleware		
	OROCOS	OpenRTM	Player
Command (asynchronous)	Operation	–	Command
Methods (synchronous)		Service port	–
Buffered/shared data (asynchronous)	Data port	Data port	Data interface
Runtime modifiable parameters	Property	Conf. interface	Service port
States of the component	Preoperational, stop, running	Created, inactive, active	–

is provided. Also, the information exchange among nodes is performed through *messages*. When many nodes are running, it is convenient to render the peer-to-peer communications as a graph. A node interested in making information accessible publishes this data via topics. When another node is interested in a certain kind of data, it will subscribe to the appropriate topic.

This paper details the code generation for a component-based platform; specifically, OROCOS platform has been selected as a target example because this is the most widespread component-based MW. As commented above, RTT provides the core of OROCOS component structure (*TaskContext* class) and how to specify the functionality of component-based applications (*Deployment* ML document). Thus, on the one hand, an OROCOS component for every UML component must be generated, and on the other hand, the logic of the application jointly with its scheduling features must be detailed in the deployment document.

Task context class, offered by RTT library, fixes the interface of OROCOS components which is defined by

- Methods required for defining the execution state of components.
- Data ports in order to provide external accessibility to components.
- Properties for configuration information can be modified in runtime.

Figure 4 illustrates the templates for generating an OROCOS header and source code.

In addition, the logic of the application and the scheduling features are in *deploymentComponent* ML file [45]. This file is the engine for OROCOS MW because it collects all necessary information for running robotic applications successfully. The main structure of this file is illustrated in Fig. 5a, which is formed by three main parts. First, the library path where OROCOS components are located is indicated. Next, the information interchanged among components has to be detailed.

Figure 5b details different examples of interchangeable information defined as connection points. For instance, *visionValueConn* is the connection point where the information captured by a camera is going to be stored.

Regarding scheduler, OROCOS offers two primitives, one for RT scheduler (ORO_SCHED_RT) and the other for the rest of the cases (ORO_SCHED_OTHERS). Figure 5c illustrates an example of deployment information for the image capturing by a camera. This is a periodic activity where a camera provides an image every 33 ms and the selected scheduler is RT.

3.3 Model to text transformation rules

This section identifies the main common transformation rules for every component-based platforms but taking OROCOS MW as an example of target platform.

- R1—rule 1 generation of application components for selected MW (*MW_App_Components*).
- R2—rule 2 provide external accessibility to *MW_App_Components*' information. Hence, the UML ports of the UML components are transformed to buffered or shared data in the case of *MW_App_Components*. R2 also processes UML interfaces to give the execution engine the communication dialogue between middleware application components.
- R3—rule 3 generation of synchronous methods and asynchronous commands for *MW_App_Components*. Thus, R3 rule is applied to every protected method of the UML class that makes possible the UML component.
- R4—rule 4 generation of runtime modifiable information of *MW_App_Components*.
- R5—rule 5 generation of MW execution engine's required information for setting-up *MW_App_Components*. To do this, R5 processes the value of the *sample* property [39]. If this value differs from zero, a periodic execution thread is generated with this period. Otherwise, a non-periodic execution thread is generated.

Fig. 4 Templates for generating the header (a) and source (b) files of OROCOS components

```

namespace GRAV{
class OrocosNameOfAtomicClass: public TaskContext public AtomicClass{
protected:
/* Ports definition */
[OutputPort | InputPort] PropertyType port_PropertyName;
public:
OrocosNameOfAtomicClass();
virtual ~OrocosNameOfAtomicClass ();
bool configureHook();
bool startHook();
void updateHook();
void stopHook();
};
}
    
```

(a)

```

namespace GRAV{
OrocosNameOfAtomicClass::OrocosNameOfAtomicClass(const std::string name)
: RTT::TaskContext(name, PreOperational),AtomicClass(){
this->addPort("label", port_PropertyName); // Publish Ports with a label
this->addOperation("label",&OrocosNameOfAtomicClass::setPropertyName, this,
RTT::OwnThread); // provide external access to protected properties
this->addProperty("label",propertyName); // add public properties
bool OrocosNameOfAtomicClass ::configureHook(){configure(); return true;}
bool OrocosNameOfAtomicClass ::startHook(){start(); return true; }
void OrocosNameOfAtomicClass ::updateHook(){
//update the data to OutputPorts
portPropertyName.write(this.getPropertyName());
// update value of the property with the value stored in inputPorts
portPropertyName.read(aux); this.setPropertyName(aux);
}
void OrocosNameOfAtomicClass ::stopHook(){stop();}
}ORO_CREATE_COMPONENT_TYPE()
ORO_LIST_COMPONENT_TYPE(GRAV::OrocosNameOfAtomicClass);
    
```

(b)

Table 6 shows the result obtained if the above-described rules are applied to the XMI file, having OROCOS as the target MW.

XSLT (XML Stylesheet Language Transformer) standard [46] offered by W3C has been used for implementing the generator. XML style sheets allow XML documents to be filtered and processed by means of templates [47]. A XML style sheet has been developed for transforming XMI.xml input file into a set of *MW_App_Components* and *DeploymentComponent.xml* file.

4 Case studies: industrial vs service robots

Robots can be classified into different categories depending on their function and the market needs they are designed for. This section presents two examples, one for every major class of robots: *industrial robots* and *service robots*. The former solves real industrial problems and work in structured environments. The latter resolves quotidian tasks in non-structured, unknown and humanized environments. These two examples have

Fig. 5 Example of deployment component file: a general overview, b connection points and c camera OROCOS component deployment information

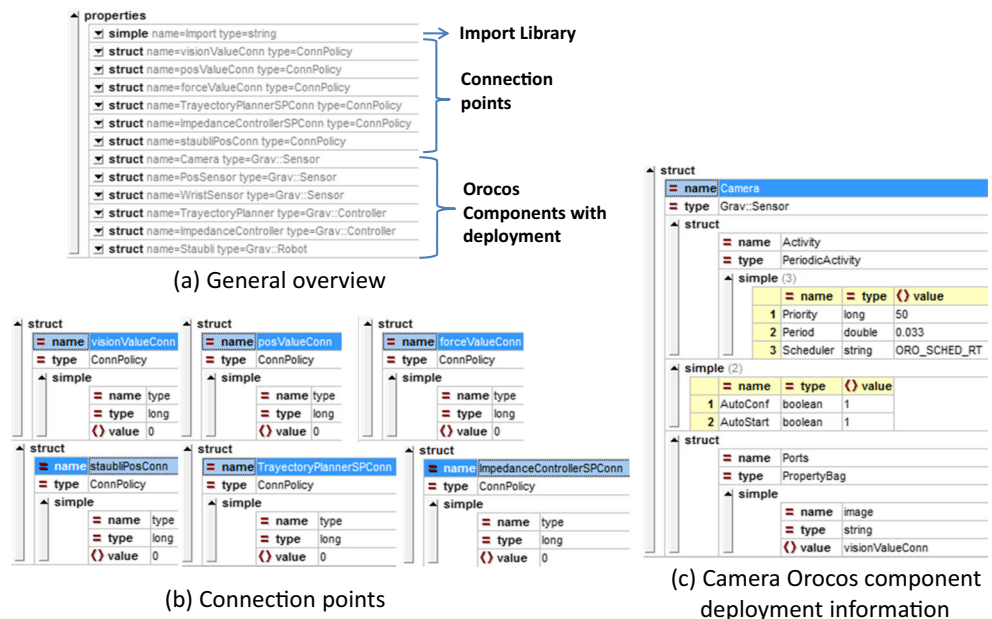


Table 6 Transformation rules for OROCOS MW

Rule	OROCOS
R1	A MW application component for each UML component. The interface of the resulting components is fixed by the communication middleware's execution engine (i.e. OROCOS RTT library provides <i>TaskContext</i> concept that fixes the interface with these six methods: <i>constructor</i> , <i>destroy</i> , <i>configureHook</i> , <i>startHook</i> and <i>stopHook</i>).
R2	UML ports transformed to buffered or shared data. This implies the generation of data ports. The R2 also processes UML interfaces to give the execution engine the communication dialogue between middleware application components. This latter is expressed in <i>OROCOS</i> in <i>deploymentComponent</i> file with <i>connectionPoint</i> concept.
R3	As a result of this transformation rule, a synchronous method for each <i>getPropertyName</i> method and an asynchronous command for each <i>setPropertyName</i> method are added to the middleware's application component.
R4	OROCOS middleware expresses these parameters as properties in a <i>deploymentComponent</i> file.
R5	To do this, it processes the value of the sample property. If this value differs from zero, a periodic execution thread is generated with this period. Otherwise, a non-periodic execution thread is generated. Related to the scheduler, if possible, R5 transformation rule provides the execution thread with a real-time scheduler (<i>ORO_SCHED_RT</i>).

been selected in order to remark the reuse of code regardless of the task and the type of robot.

4.1 Industrial application: the assembly of the headlamps

Nowadays, a vehicle headlamp is a highly sophisticated device that has to pass exhaustive quality inspections, normally demanded by the car manufactures [48]. One of the stages of the production process of headlamps is the assembly of the components where the main operation consists, basically, of the positioning and fixing of the lens—made of polycarbonate—over a black housing made of polypropylene (Fig. 6). The rest of the components: reflector, lighting system and bezel, are placed into the housing.

Because of the nature of the production process, the dimensional variability of both housing and lens is relatively high if it is compared with the position

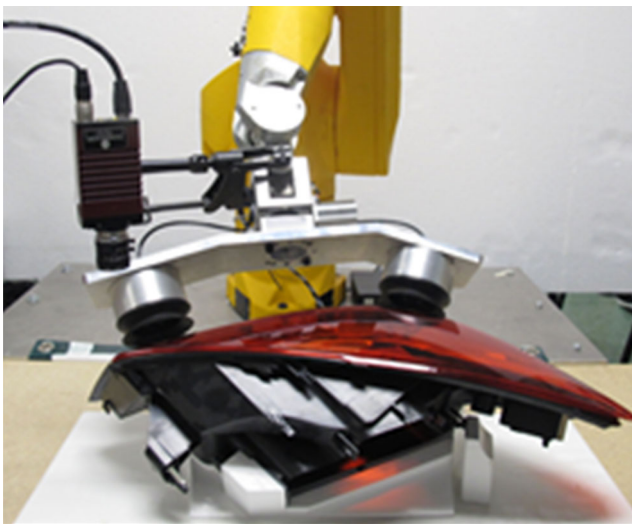


Fig. 6 Photograph of the assembly carried out by the robot

requirements demanded by the car manufacturers. Obviously, this dimensional variation supposes a problem during the assembly.

The experimental setup, implemented using the methodology developed in this paper, is the assembly of both components using an industrial manipulator [49].

The idea is, considering the housing as a fixed element and using its gum channel, to move the lens inside the channel with the robot—whose width is around 2 mm—in order to minimize the contact forces. For this assembly task, two sensors were used: a wrist force sensor attached to robot tip, which can determine the forces and torques generated by the manipulator and its contact point; and a vision sensor whose mission, jointly with image-processing algorithms, is to determine the position of the gum channel. So, the assembly procedure is as follows: once the gum channel is identified, the manipulator moves the lens to the housing channel, and then, using the wrist sensor, the contact point is determined, together with the forces and torques exerted by the lens over the housing. Finally, the robot goes to the position that minimizes the forces and torques. An impedance algorithm has been applied as a force controller. Hence, this behaviour is illustrated in Fig. 7a with a UML component diagram. The selected HW/SW platform is detailed in Fig. 7b. In this case, the OROCOS communication middleware has been selected.

The source code generator (M2T transformer) processes the corresponding XMI file and generates the application OROCOS components. Figure 8 illustrates, as an example, the source for Guppy80 OROCOS component.

Afterwards, these components are compiled with a *makefile* mechanism generating a library. Additionally, the M2T transformer also generates the corresponding

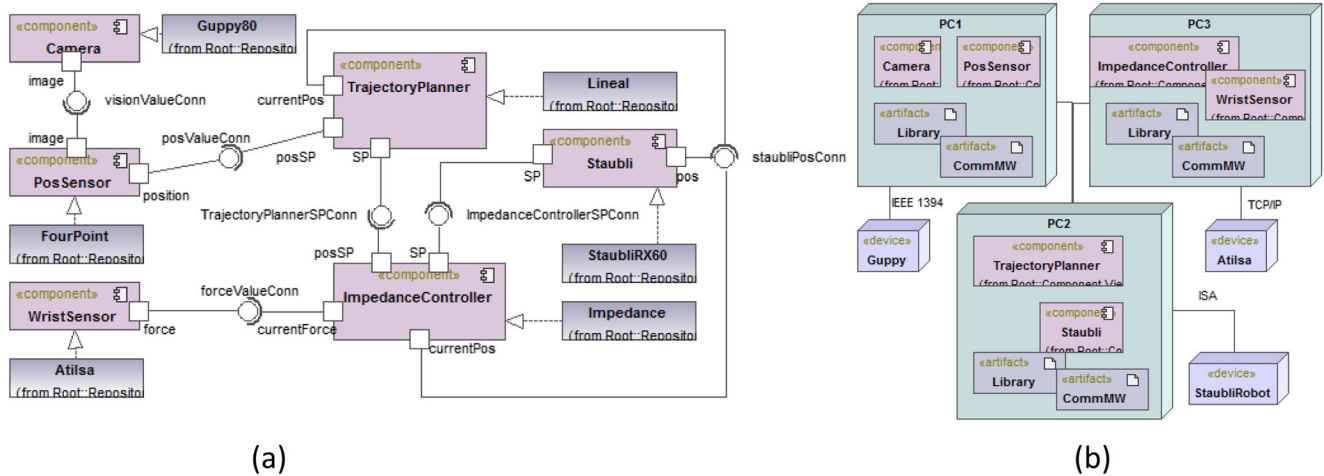


Fig. 7 Assembly of the headlamps functionality (a) and platform (b) modelling

deploymentComponent XML file for this robotic task. As mentioned above, Fig. 5a shows the three main parts of the XML file. Firstly, the path to import the previously obtained library, after the connection points and, secondly, the deployment information for every OROCOS component is indicated.

Specifically, six connection points have been generated, one for every UML interface of the application logic (see Fig. 7a). Figure 5b details the characterization of the connection points defined for this case study. All of them behave as a shared long data type (*value=0* and *type=long*). Figure 5c illustrates the deployment information for Guppy component. The execution thread information defines a periodic activity with *priority=50*, *period=0.003* and RT scheduler. *Autoconf* and *Autostart* features are active, so the OROCOS task context invokes automatically *configureHook* and *startHook*

methods. The output data port is connected to the corresponding connection point.

Finally, the real-time application starts running under OROCOS MW with *deployer-xenomai -s deploymentComponentFileName.xml* command.

4.2 Service application: the cleaning of a flat surface

The cleaning of flat surfaces such as windows, walls and tables are quotidian tasks that humans perform simply. Nevertheless, these tasks are not trivial for service robots because they need to maintain a constant contact with the surface to be cleaned without damaging this surface. This case study describes how a humanoid robot can perform a cleaning task by applying an impedance control in order to orient the side of the robot’s hand normally to the surface of a table. Besides,

Fig. 8 Source code of Guppy80 OROCOS component

```

namespace GRAV{
OrocosGuppy80::OrocosGuppy80(const std::string name): RTT::TaskContext(name,
PreOperational),Guppy80(){
    this->addPort("image", portImageRaw);
    this->addPort("imageColor", portImageColor);
    this->addPort("imageGray", portImageGray);
    this->addOperation("setWidth",&OrocosGuppy::setWidth, this, RTT::OwnThread);
    this->addOperation("getWidth",&OrocosGuppy::getWidth, this, RTT::OwnThread);
    this->addOperation("setHeight",&OrocosGuppy::setHeight, this, RTT::OwnThread);
    this->addOperation("getHeight",&OrocosGuppy::getHeight, this, RTT::OwnThread);
    this->addOperation("setGain",&OrocosGuppy::setGain, this, RTT::OwnThread);
    this->addOperation("getGain",&OrocosGuppy::getGain, this, RTT::OwnThread);
    this->addOperation("setShutter",&OrocosGuppy::setShutter, this, RTT::OwnThread);
    this->addOperation("getShutter",&OrocosGuppy::getShutter, this, RTT::OwnThread);
}
bool OrocosGuppy80::configureHook(){configure(); return true;}
bool OrocosGuppy80::startHook(){start(); return true;}
void OrocosGuppy80::updateHook(){
    portImageRaw.write(this.getImageRaw()); portImageColor.write(this.getImageColor());
    portImageGray.write(this.getImageGray());
}
void OrocosGuppy::stopHook(){stop();}
}ORO_CREATE_COMPONENT_TYPE() ORO_LIST_COMPONENT_TYPE(GRAV::OrocosGuppy80);
    
```

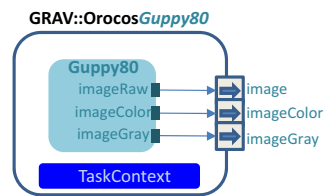
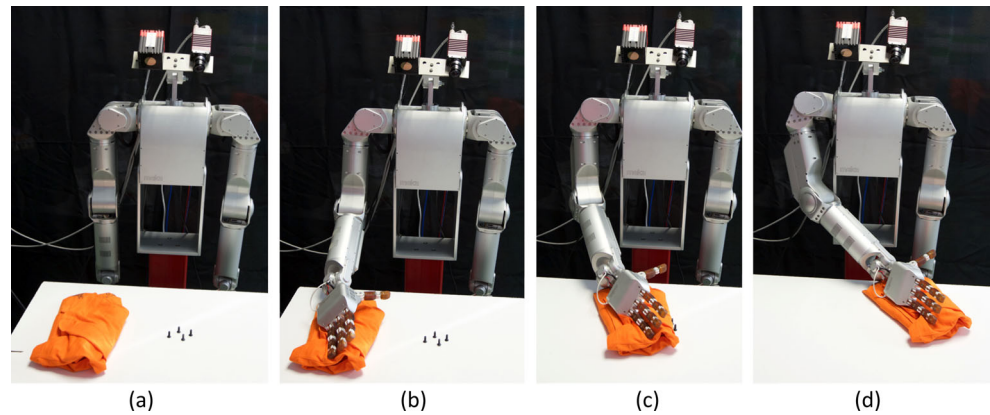


Fig. 9 Photograph of the wiping carried out by the Meka robot: **a** the positions of screws and cloth are addressed, **b** the Meka manipulator moves the arm to the cloth and **c, d** robot follows a trajectory generated by a trajectory planner algorithm



a computer vision system is required for locating the particles that should be wiped from the table.

The experimental setup, that has been implemented using the methodology developed in this paper, is the wiping of a flat surface such as a table with the Meka humanoid robot.

The goal of this task is to remove a set of screws placed on a fixed and flat surface. This implies not only the location of the screws to be removed but also the contact force has to be controlled for not damaging the surface.

For this cleaning task, two sensors were used: a force sensor attached to robot wrist, which can determine the forces and torques generated by the manipulator; and a vision sensor whose mission, jointly with image-processing algorithms, is to determine the position of the screws on the table surface. Hence, the cleaning process is the following: first, the positions of screws and cloth are addressed (Fig. 9a). After, the Meka manipulator moves the arm to the cloth (Fig. 9b), so at this moment, the robot is ready to start the *cleaning process*. It is worth noting, though, that making use of the information provided by the ATI force sensor and an impedance algorithm as force controller, the forces and torques exerted by the hand are known. Finally, as the table is a flat surface, robot follows a trajectory generated by a trajectory planner algorithm

(Fig. 9c, d). This trajectory is generated from the centre gravity of the cloth and the screws to be cleaned.

Hence, this behaviour of the cleaning process itself is illustrated in Fig. 10a. The selected HW/SW platform is detailed in Fig. 10b.

Figure 11 illustrates the source code for managing GX1050 Prosilica camera in OROCOS middleware.

As commented above, M2T generator finishes with the generation of the corresponding *deploymentComponent* XML file for this robotic task.

Figure 12a shows the three main parts of the XML file. Concretely, six connection points have been generated, one for every UML interface of the application logic. Figure 12b details the characterization of two connection points for this case study. The former contains the image captured by the camera, and the latter the position of the screws to be cleaned. Figure 12c illustrates the deployment information for GX1050 Prosilica and image-processing OROCOS components. In both cases, the execution thread information defines a periodic activity with priority=50, period=0.05 and no real-time scheduler. *Autoconf* and *Autostart* features are active, so the OROCOS task context invokes automatically *configureHook* and *startHook* methods.

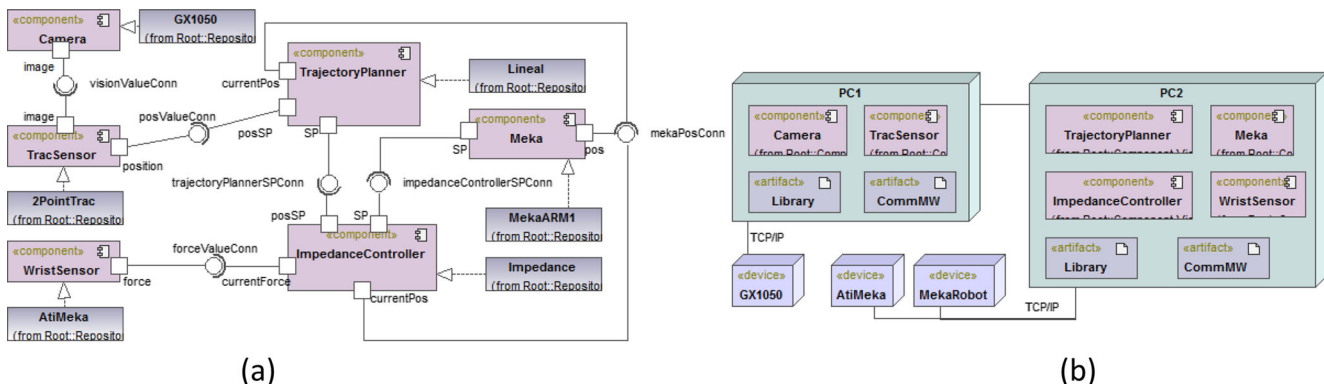


Fig. 10 The cleaning flat surface process: functionality (a) and platform (b) modelling

Fig. 11 Source code of GX1050 OROCOS component

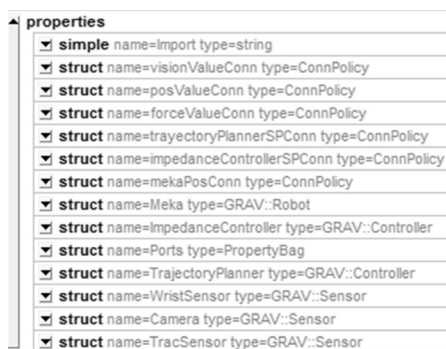
```

namespace GRAV{
OrocosGX1050::OrocosGX1050(const std::string name): RTT::TaskContext(name,
PreOperational),GX1050(){
    this->addPort("image", portImageRaw);
    this->addPort("imageColor", portImageColor);
    this->addPort("imageGray", portImageGray);
    this->addOperation("setWidth",&OrocosGuppy::setWidth, this, RTT::OwnThread);
    this->addOperation("getWidth",&OrocosGuppy::getWidth, this, RTT::OwnThread);
    this->addOperation("setHeight",&OrocosGuppy::setHeight, this, RTT::OwnThread);
    this->addOperation("getHeight",&OrocosGuppy::getHeight, this, RTT::OwnThread);
    this->addOperation("setGain",&OrocosGuppy::setGain, this, RTT::OwnThread);
    this->addOperation("getGain",&OrocosGuppy::getGain, this, RTT::OwnThread);
    this->addOperation("setShutter",&OrocosGuppy::setShutter, this, RTT::OwnThread);
    this->addOperation("getShutter",&OrocosGuppy::getShutter, this, RTT::OwnThread);
    this->addProperty("ip",ip);
}
bool OrocosGuppy80::configureHook(){configure(); return true;}
bool OrocosGuppy80::startHook(){start(); return true;}
void OrocosGuppy80::updateHook(){
    portImageRaw.write(this.getImageRaw()); portImageColor.write(this.getImageColor());
    portImageGray.write(this.getImageGray());
}
void OrocosGuppy::stopHook(){stop();}
}ORO_CREATE_COMPONENT_TYPE() ORO_LIST_COMPONENT_TYPE(GRAV::OrocosGX1050);
    
```

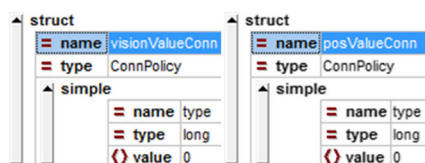
5 Conclusions

This paper presents a framework that supports the development cycle of robotic arm tasks. The model-driven engineering approach has been followed to provide support to the

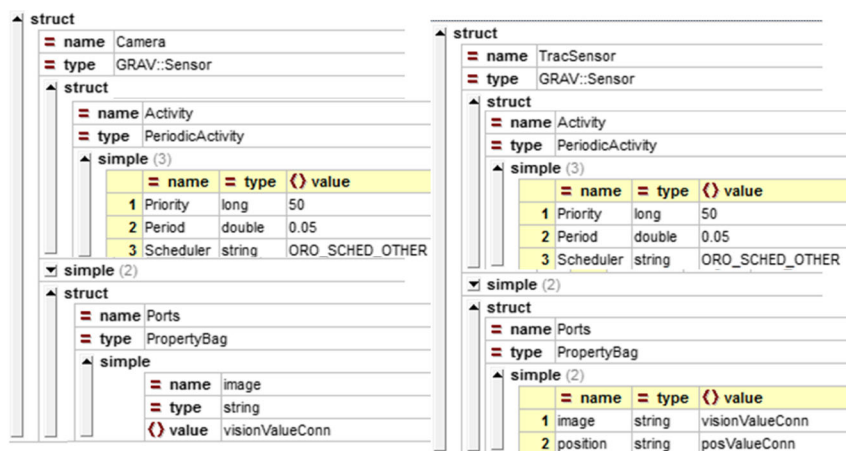
design and coding phases of the development cycle. The model concept has been used for describing the functionality of the robotic arm task, which takes place during the design phase. Also, M2T transformation has been identified to generate the target code for component-based communication framework.



(a) General overview



(b) Connection points



(c) Camera and position OrocOS components

Fig. 12 Deployment XML file for screws cleaning robotics task: **a** general overview, **b** connection points and **c** camera and position of OROCOS components

The authors' proposed framework uses UML as a modeling language. Specifically, UML 2.x component and deployment diagrams have been used. The functional model is the input model to M2T transformer that generates the target code for communication middleware. As XMI is a standard notation, the proposed framework is valid for any UML modelling tool that supports this import/export option. Furthermore, it is important to note that this framework is compatible with the general robot control system because it is based on widespread standards.

Finally, the proposed modelling approach has been validated with two case studies, one for an industrial robotics and other for service robotics. These two examples are very representative in order to note the achievement of the reusability feature regardless of the task but also type of robot.

Acknowledgments This work was financed in part by the MCYT&FEDER under DPI2011-27284 and by Andalusia Government under AGR-6429.

References

- Gil P, Pomares J, Puente ST, Candelas FA, Garcia GJ, Corrales JA, Torres F (2009) A cooperative robotic system based on multiple sensors to construct metallic structures. *Int J Adv Manuf Technol* 45(5):616–630
- Edsinger A (2007) Robot manipulation in human environments. Ph.D. Dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science
- Marcos F, António Paulo M, Pedro N (2012) A low-cost laser scanning solution for flexible robotic cells: spray coating. *Int J Adv Manuf Technol* 58:103–1041
- Mendes N, Neto P, Simão MA, Loureiro A, Pires JN (2014) A novel friction stir welding robotic platform: welding polymeric materials. *Int J Adv Manuf Technol*. DOI: 0.1007/s00170-014-6024-z. [Online published]
- Dragan M, Milos G, Nikola S, Zoran D, Sasa Z, Branko K, Ljubodrag T (2011) Reconfigurable robotic machining system controlled and programmed in a machine tool manner. *Int J Adv Manuf Technol* 53:1217–1229
- Chella A, Cossentino M, Gaglio S, Sabatucci L, Seidita V (2010) Agent oriented software patterns for rapid and affordable robot programming. *J Syst Softw* 83(4):557–573
- Iborra A, Caceres DA, Ortiz FJ, Franco JP, Palma PS, Alvarez B (2009) Design of service robots, experiences using software engineering. *IEEE Robot Autom Mag* 16(1):24–33
- Wahl FM, Kroger T (2009) *Advances in robotics research: theory, implementation, application*. Springer-Verlag Berlin and Heidelberg GmbH & Co. K
- Lihui W, Bernard S, Mohammad G, Göran A (2014) Robotic assembly planning and control with enhanced adaptability through function blocks. *Int J Adv Manuf Technol*. doi:10.1007/s00170-014-6468-1 [Online published]
- Heineman GT, Council WT (2001) *Component-based software engineering: putting the pieces together*. Addison-Wesley
- Sommerville I (2007) *Software engineering, eight edition*, Pearson Education
- Brooks A, Kaupp T, Makarenko A, Williams S, Oreback A (2005) Towards component-based robotics. *Proc IEEE Int Conf Intell Robot Syst (IROS)* pp: 163–168
- Brugali D, Scandurra P (2009) Component-based robotic engineering (part I) reusable building blocks. *IEEE Robot Autom Mag* 16(4):84–96
- Brugali D, Shakhimardanov A (2010) Component-based robotic engineering (part II) systems and models. *IEEE Robot Autom Mag* 17(1):100–112
- Gamez J, Robertsson A, Gomez Ortega J, Johansson R (2008) Sensor fusion for compliant robot motion control. *IEEE Trans Robot* 24(2):430–441
- Selic B (2003) The pragmatics of model-driven development. *Softw IEEE* 20(5):19–25
- Streitferdt D, Wendt G, Nenninger P, Nyßen A, Lichter H (2008) Model driven development challenges in the automation domain. Annual IEEE International Computer Software and Applications Conference. Turku, Finland
- Balasubramanian K, Gokhale A, Karsai G, Sztipanovits J, Neema S (2006) Developing applications using model-driven design environments. *Computer* 39(2):33–40
- Schmidt D (2006) Guest editor's introduction: model-driven engineering. *Computer* 39(2):25–31
- Sharygina N, Browne JC, Kurshan RP (2001) A formal object-oriented analysis for software reliability: design for verification. *Lect Notes Comp Sci Fundam Approaches Softw Eng* 2029:318–332
- Arun Kumar R, Bruno M, Adriana T (2014) Solution space modeling for robotic systems. *J Softw Eng Robot (JOSER)* 5(1):89–96
- Geisinger M, Barner S, Wojtczyk M, Knoll A (2009) A software architecture for model-based programming of robot systems. *Lect Notes Comput Sci Adv Robot Res* pp. 135–146
- Alonso D, Vicente-Chicote C, Ortiz F, Pastor J, Álvarez B (2010) V3CMM: a 3-view component meta-model for model-driven robotic software development. *J Softw Eng Robot* 1(1):3–17
- SmartSoft MDS Toolchain (2013) SmartSoft model driven development software design toolchain, [Online] Available at: <http://smart-robotics.sourceforge.net/index.php>
- Christian S, Alex L, Matthias L, Dennis S, Inglés-Romero JF, Cristina V-C (2013) Model-driven software systems engineering in robotics: covering the complete life-cycle of a robot. Workshop Roboter-Kontrollarchitekturen, Informatik 2013. Springer LNI der GI, Koblenz, pp 2780–2794
- Schlegel C, Steck A, Lotz A (2012) Robotic software systems: from code-driven to model-driven software development. *Robot Autom Robot Syst Appl Control Program Intechopen* pp:473–502
- Schlegel C, Steck A, Lotz A (2012) Model-driven software development in robotics: communication patterns as key for a robotics component model. In: *Introduction to modern robotics*. iConcept Press
- Zahavi R (2000) *Enterprise application integration with CORBA*. Wiley, New York
- Brugali D, Gherardi L, Luzzana A, Zakharov A (2012) A reuse-oriented development process for component-based robotic system. In: *Proc. of the 3rd International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAN)*
- BRICS-Best Practice in Robotics Project, [Online] Available at: <http://www.best-of-robotics.org>
- Bruyninckx H (2001) Open robot control software: the OROCOS project. In: *Proc IEEE Int Conf Robot Autom (ICRA)*, pages 2523–2528. Seoul, Korea
- Russell J, Cohn R (2012) ROS (robotic operating system), VSD
- Miller J, Mukerji J (2001). Model driven architecture (MDA). OMG, ormsc/2001-07-01, Architecture Board ORMSC1, July 2001

34. Booch G, Rumbaugh J, Jacobson I (2005) The unified modeling language user guide, 2nd Edition, Addison-Wesley Professional
35. Jones L, Fowler J, James S, Fu Y (2012) UML based design of LEGO Robots. Proc Int Conf Softw Eng Res Pract pp:10-16
36. Layne A, Mason A, Fu Y, Wagaw M (2012) UML model based design of the claw car robot. Proc Int Conf Softw Eng Res Pract pp:3-9
37. Kim M, Kim S, Park S, Choi M-T, Kim M, Gomaa H (2008) UML-based service robot software development: a case study, *Advances in service robotics*, Ahn HS (ed.), ISBN: 978-953-7619-02-2, InTech, DOI: [10.5772/5947](https://doi.org/10.5772/5947)
38. OMG. Meta Object Facility (MOF) 2.x XMI mapping specification. [Online] Available: <http://www.omg.org/spec/XMI/> , Last access in March 2014
39. Sanchez Garcia A, Estevez E, Gomez Ortega J, Gamez Garcia J (2013) Component-based modelling for generating robotic arm applications running under OROCOS middleware. Proc IEEE Int Conf Syst Man Cybern pp: 3633-3638
40. Salmini A, Tomba F (2011) *Communicating with XML*. Springer, New York
41. Deliverable D-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 . Last Access in May 2015
42. OPENRTM [Online] Website: <http://www.openrtm.org/openrtm/en/node/780> . Last Access in May 2015
43. Gerkey B, Vaughan R, Howard A (2003) The player/stage project: tools for multi-robot and distributed sensor systems. In Proc. of the International Conference on Advanced Robotics
44. ROS-INDUSTRIAL, [Online] Website: <http://rosindustrial.org/>. Last Access in May 2015
45. OROCOS—the deployment component (2012). [Online]. <http://www.orocos.org/stable/documentation/ocl/v2.x/docxml/orocos-deployment.html> . Last Access in May 2015
46. Tidwell D (2001) *XSLT*, Ed. O'REILLY
47. Estévez E, Marcos M, Orive D (2007) Automatic generation of PLC automation projects from component-based models. Int J Adv Manuf Technol 35(5–6):527–5440
48. 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. Int J Adv Manuf Technol 68(5-8):1123–1136
49. Gomez Ortega J, Gamez Garcia J, Satorres-Martínez S, Sanchez Garcia A (2011) Industrial assembly of parts with dimensional variations. Case study: assembling vehicle headlamps. Robot Comput Integr Manuf 27(6):1001–1010