

# Automatic transformation tools of UML design models from virtual prototypes of multi-jointed robots

Hyun Seung Son<sup>1</sup> · R. Young Chul Kim<sup>1</sup>

Received: 3 June 2016 / Revised: 14 September 2017 / Accepted: 26 December 2017 /

Published online: 11 January 2018

© Springer Science+Business Media, LLC, part of Springer Nature 2018

**Abstract** Most of robotic companies develop a control programming of multi-jointed robots, which spend too much time to manually adjust the moving functions of the robots. To solve this problem, we adapt the virtual prototyping (VP) to develop the control program of the robotic behaviors. For software engineers, in order for them to easily program this robot, we also apply metamodel mechanism to convert UML models with virtual prototyping model. We propose the automatic model transformation from the virtual prototyping model to UML models, which will then develop coding based on UML models. To prove our mechanism's efficiency, we implement *Robot to UML Translator (RUT)* as our transformation rules with ATLAS transformational language. Lastly, we show experimental validation about the consistency of our proposed technique with an example of multi-jointed robot prototype models.

**Keywords** Model transformation · UML · Virtual prototyping · Software controller · Multi-jointed robot · Virtual Robert (VirRobot)

## 1 Introduction

As an example of six-legged and 18 jointed robot [1], the software controller must coordinate 18 separate motors in a precise and timely manner. But it is difficult to manually program the behavior of the multi-jointed robot. For this reason, we suggest that virtual prototyping models [2, 3] are used to simulate the robotic behavior to operate more precise controls. Our idea develops a virtual model before constructing the robot. In the virtual prototype model, we must consider whether a given robot design will work effectively or not. This information makes it

---

✉ R. Young Chul Kim  
[bob@selab.hongik.ac.kr](mailto:bob@selab.hongik.ac.kr)

Hyun Seung Son  
[son@selab.hongik.ac.kr](mailto:son@selab.hongik.ac.kr)

<sup>1</sup> SE Lab, Department of Computer and Information Communication, Hongik University, Sejong 30016, Republic of Korea

possible to obtain invaluable data about the robotic behavior prior to its actual construction. The actual development of a robot's control software generally requires a redesigning of the software itself with UML. However, if UML design models can be generated automatically from virtual prototype models, the production costs will vastly reduce.

This paper introduces an innovative technique to automatically generate an UML-based model from a virtual prototype-based model. At first, for the model transformation, a common meta-model for both the virtual prototype and the UML models must be defined. In this study, a Meta-Object Facility (MOF) [4] of Model Driven Architecture (MDA) [5] is used to define two common meta-models. Since the Object Management Group (OMG) defined the meta-model of the UML sample using an MOF, we also define the meta-model of the virtual prototyping model, VirRobot [6], with an MOF as well. Secondly, in order to successfully transform one model to the another, most effective model transformation language is required. A number of studies have been recently worked on model transformation languages, such as UML Model Transformation (UMT) [7], Model Transformation Language (MTL) [8], Query / View / Transformation (QVT) [9] and ATLAS Transformation Language (ATL) [10, 11]. For the purpose of this research, ATL has been selected as the most viable option for the model transformation languages on the basis of transformation engine of the MDA framework. This is easy to use and define complicated transformation rules. In addition, it is well embedded in the JAVA development environment, Eclipse, by using this option. Thus, we can create eight ATL rules to transform a VirRobot virtual prototype model into UML-based class diagram while simultaneously developing an automatic transformation method in JAVA.

It is worth mentioning that UML 2.2 is composed of 14 language units and 39 packages. For the purpose of this study, our technique only utilizes a class diagram. To clarify the process, we extract a subset from the UML meta-model, which is pertinent to the automatic transformation. Since the VirRobot prototyping model contains information, the class diagram is sufficient as a design model on the robot's static structure and basic operations. In the future, we plan to extend the VirRobot to provide a virtual simulation function. This will require a class diagram, state diagram, and a sequence diagram.

The paper is organized as follows: Section 2 describes the virtual prototyping process, the VirRobot prototyping model, and the ATL transformation sequence. Section 3 briefly describes the overall transformation process from a virtual-based to a UML-based model. In Sections 4 and 5 we will provide detailed explanations about the transformation steps. In Section 6 we will review our experiment for results of the three multi-jointed prototype robot models, and compare the simulated UML models with the actual models which are manually generated by experts. In Section 7, we will review related studies from the field. Lastly, we will provide the concluding comments and suggestions for further research.

## 2 Related works

Virtual prototyping [12] is used to validate a design before committing precise time and resources to its actual physical construction. It includes creating on computer generated 3D shapes and combining them with each other. Then, different mechanical motions are tested. We either individually or collectively observe the various stresses of the product that may encounter in the real world.

In virtual prototyping, a product data model is employed to build a computational prototype. Model operations and analysis are performed to reflect physical representations of real

world situations. The virtual prototyping-aided design is based on the integration of computer supported modeling and virtual reality interfacing as shown in Fig. 1. Product realization activities are first performed with respect to the virtual world, where all the necessary product data and manufacturing processes are modeled.

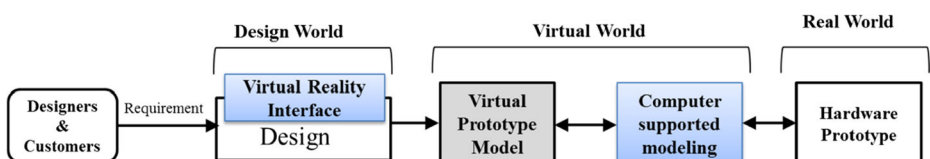
It is important to note that virtual prototype models are only used to validate a design before creating an actual software artifact. Also the development of real artifacts requires an overall redesign of the models themselves. For example, Microsoft Robotics Studio (MSRS) [13] can develop a robot model in a virtual environment, but the results are limited to simulation purposes only.

Numerous studies have been conducted in the designing of different types of multi-jointed robots. One such design is LAURON [14], an insect-like walking robot used for transporting heavy loads on a sandy terrain. Its controlled movement allows an autonomic walking behavior even in rough terrain. Another design is the AirBug [15], which has a insect-like six-legged with pneumatic muscles. It focuses on the control concept of the antagonistic actuators. Other designs include Climber [16], a six-legged climbing robot used for high payloads, and Dante II [17], an eight pantographic-legged robot used to explore inside volcanic craters.

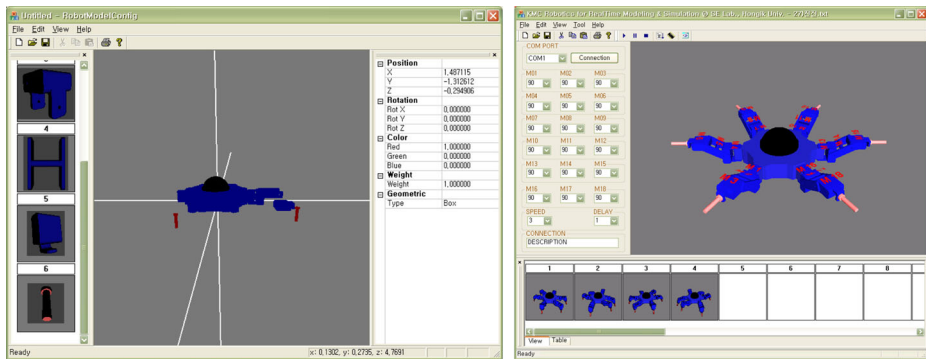
VirRobot [18] is a virtual prototyping tool-set for multi-jointed robots, developed by Hongik University and KMC Robotics Corp. in Korea. It is currently used as a form of robotic training equipment in many domestic schools. The VirRobot can design a multi-jointed robot consisting of several legs (up to 10) with a camera and multiple sensors to detect ultrasonic waves and temperatures. Furthermore, the VirRobot provides both model combination and motion generation facilities. Figure 2(a) describes the model combination tool of the VirRobot. This tool combines parts such as the legs and the joints into a specific multi-jointed robot of virtual medium. Each part has its own operational limitations. Moreover, the model combination tool stores the virtual robot model in an XML format in order to communicate with the motion generation tool.

The motion generation tool produces motions for the virtual prototyping model. As described in Fig. 2(b), it performs when simulated in a virtual environment. This process is visible on screen, which allows the developer to view and select varying operational angles on all joints. For each operation, the ‘moving forward’ or ‘moving backward’, we can assign and simulate a series of motions for all multi-jointed legs involved. This tool stores an XML file containing information about the robot’s parts, joints, and library APIs. Along with all possible motions and another view of the virtual prototype model itself. Furthermore, the XML output makes it possible to generate a UML-based class diagram along with source code programs at a later stage.

ATLAS Transformation Language (ATL) [19] is a model transformation language (MTL). MTL is capable of translating an original model into several other models. It provides declarative and imperative transformations. Mostly, the preferred transformation writing style is declarative. However, the imperative transformation constructs with Object Constraint Language (OCL) [20]. This is used to specify the mappings that are too complex to be handled declaratively. An ATL transformation program is composed of rules that define how source model elements are matched



**Fig. 1** The principle of virtual prototyping-aided design



(a) The model combination tool

(b) The motion generation tool

**Fig. 2** The virtual prototyping tool named VirRobot

and navigated in order to create and initialize the elements of the target models. There is an associated ATL development toolkit plug-in available on an open source from the Eclipse Modeling Framework (EMF) that implements the ATL transformation language.

### 3 Our robot software development process

This section describes a virtual prototyping oriented software development for multi-jointed robots, which is adapted to an UML model translator proposed in RUT. This follows the waterfall model of software development processes. It is composed of four phases: requirements analysis, prototyping, design, and implementation. A testing phase is not included beyond the scope of this paper.

During the requirement analysis phase, the robot software requirement specifications are first defined. For this study, the use case method is applied to write these requirements. During the prototyping phase, a virtual robot model and all of its possible motions are developed. For this experiment, the virtual prototyping and the VirRobot [18] are used. The virtual robot model is then automatically translated into UML models during the design phase. For this, we develop transformation rules with ATL in an Eclipse environment, which allows the automatic transformation process to produce UML class diagrams from the virtual prototype model. It is worthwhile expanding this procedure to include sequence diagrams and state diagrams in the future. During the implementing phase, the executable machine code is automatically generated from the UML diagrams with our ‘Hongik MDD based Embedded S/W Component Development Methodology’ (HIMEM) tool. Figure 3 describes the entire development process and supporting tools, that is, our HIMEM tool.

Our automatic model transformation method receives input in XML from the robot prototyping model before translating it into class diagrams in XMI [21] using an ATL translator. The ATL translator requires a meta-model for each diagram in order to translate one into another. While the class diagram in UML has a standard meta-model which is defined in the OMG’s Meta Object Facility (MOF), the robot’s prototyping model needs to be defined from the bottom up. Although the UML-based meta-model covers all UML 2.2 constructs comprehensively, only some portions of these constructs are needed for the automatic ATL-based transformation. Further details will be provided later in this paper.

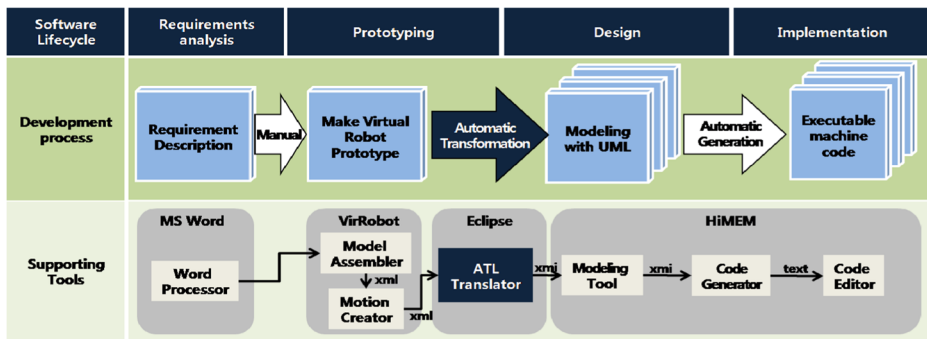


Fig. 3 The robot software development process using virtual prototyping

A VirRobot prototyping tool is used to develop a virtual prototype model during the prototyping phase. It is performed an automatic transformation of a VirRobot model into an UML-based design model with the ATL within an Eclipse environment. During the design and implementation phases, it applied with a development method and tool-set that we previously developed, that is, *Hongik MDD based Embedded S/W Component Development Methodology* (HiHEM) [22, 23]. The HiHEM mechanically generates the executable machine code from the UML-based design model. In the following sections, we provide further clarification of this proposed transformation technique and the corresponding developmental process.

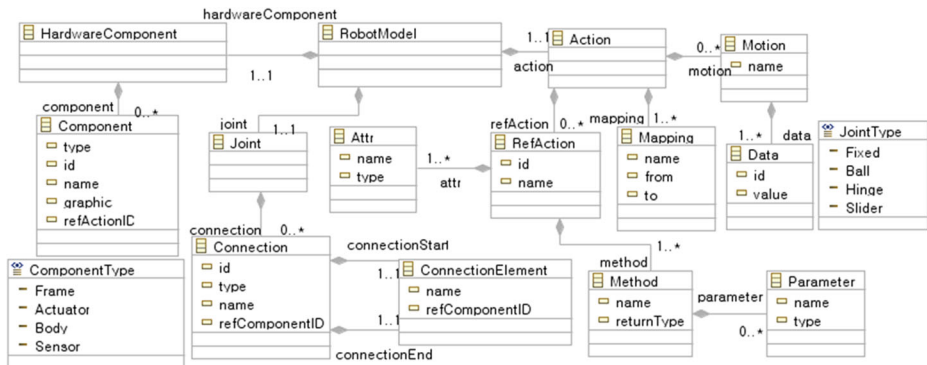
## 4 Prototyping the robot model

In this approach, we first adapt the virtual prototyping and the meta-model mechanism to the robot software development for KMC robotic company in Korea. This section explains the virtual prototype model developed using the VirRobot prototyping tools. The virtual model of a multi-jointed robot and all its possible motions are created in the prototyping phase. The robot model consists of a robot structure, hardware components, joints, motion data, and robot APIs for operations. The VirRobot generates a robot model through these steps which allow the robot model to be constructed with hardware parts, thereby connecting each component parts, and generating possible robotic motions. The VirRobot model and its operational data are then stored in XML.

It is important to note that the meta-model and the VirRobot prototyping model must be defined in order to automatically transform the robot model into a UML-based design model. This is because the ATL translator needs a meta-model from both the source and the transformed models based on the OMG group.

### 4.1 Meta-modeling for the robot model

The Meta-model of a virtual robot model should represent the actual robot model's structure and operations. Figure 4 shows the meta-model of the VirRobot model developed in this study. The component of *Robot Model* corresponds to each robot model. It is also composed of 3 components: *HardwareComponent*, *Joint*, and *Action*. For this experiment, different types of components and joints are used and defined as well.



**Fig. 4** The meta-model of the robot model

First, the *HardwareComponent* is defined with the robot's hardware parts, which can be classified into 4 types: *Frame*, *Actuator*, *Body*, and *Sensor*. The *Frame* is a hardware component with no action. It is defined using only 3D graphic data. The *Sensor* is used to produce or accept *Active* and *Passive* events. The former makes actively it interrupt, but the latter is accepted by the sensor only when receiving a request from others. For the *Actuator*, such as the DC Motor or Servo Motor, various levels of operation and speed can be assigned. The *Body* is a Micro Control Unit (MCU) controlling the robot. The *Body* is part of the robot's hardware frame and has a MCU inside the robot, that is, being controlled by the body. Furthermore, each hardware component has a *refActionID* that can be associated with an API, necessary for robotic operations.

Secondly, the *Joint* is defined as a connection of the hardware components. In this study, the meta-model has 4 types of *Joint*: *Fixed*, *Ball*, *Hinge*, and *Slider*. The *Fixed* joint type connects hardware components firmly together with no rotation or movement. The *Ball* joint provides a 360-degree rotation of the attached components. The *Hinge* joint type allows for a limited 180-degree horizontal movement only. The *Slider* can move vertically in both directions. Each joint consists of a *connectionStart*, a *connectionEnd*, and a *connection*. This means that the hardware components are connected with each other at their respective joint. Additionally, the hardware components are referred to their respective *refComponentID*.

The third element composing the *RobotModel* is the *Action* component. The *Action* is composed of *RefAction*, *Mapping*, and *Motion*. *RefAction* is a set of APIs for operating the robot. It is composed of several attributes *Attr* and functions *Method*. The *Mapping* function maps the *RefAction* APIs in relation to each other, while the *Motion* function stores information about the joints including their angle and direction of movement.

It is important to note that the meta-model of the VirRobot prototyping model defined above can also be sufficiently applied to other models when mapping and designing other multi-jointed robots. Examples of this method’s applicability are provided in the next section.

## 4.2 Constructing the virtual robot model

The virtual robot model, that is, the VirRobot model, is produced using the VirRobot prototyping tool which is based on the meta-model defined in the previous subsection. Furthermore the detailed definitions of the VirRobot model from its hardware structure to its software APIs, are provided below.

#### 4.2.1 The structure of the robot model

As the example provided in Fig. 5, the VirRobot enables us to virtually simulate a real multi-jointed robot. The robot below has six legs and 18 joints. Each equipped with a motor allows it move and rotate up to 180-degrees. Although the six legs helps the robot maintain its balance better, it allows movement to be more flexible. On fluidly performed when compared to the 2 legged versions, it creates certain problems in term of leg control and coordination.

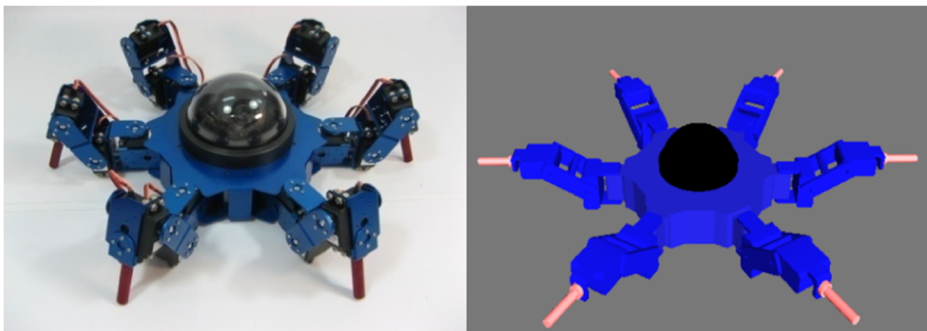
Figure 6 shows the hardware structure of the multi-jointed robot model depicted in Fig. 5. The *Head* describes the half-rounded top in the middle of the robot. The *Body* refers to the trunk to which the head and legs are connected. Each part is connected with a *Joint*. For the robot model featured in Fig. 6, only *Fixed* and *Hinge* joints are present. To apply this sample to our virtual representation, the *Frame* is used to model the legs. Each frame has its own labeling number. For example, the six-legged model has 24 *Frames*. The *Frames* numbered 4, 8, 12, 16, 20, and 24 at the bottom of the chart denote the end of each leg in Fig. 6, which are shown as red cylinders in Fig. 5.

From the robot model structure given in Fig. 6, we can validate the robot meta-model defined in section 4.1. In fact, the entire model corresponds to the *RobotModel* of the meta-model described in Fig. 4. First, the *Component* (the top left side of Fig. 4) composes the *HardwareComponent* which maps the *Frame* (in Fig. 6). Next, the meta-model defines that each *Joint* has a *ConnectionStart*, a *Connection*, and a *ConnectionEnd*. For each *Joint*, on either *Fixed* or *Hinge*, we can see that two components are connected to each side. Figure 7 provides specific mapping of each element of the robot meta-model along with corresponding elements in the actual robot model.

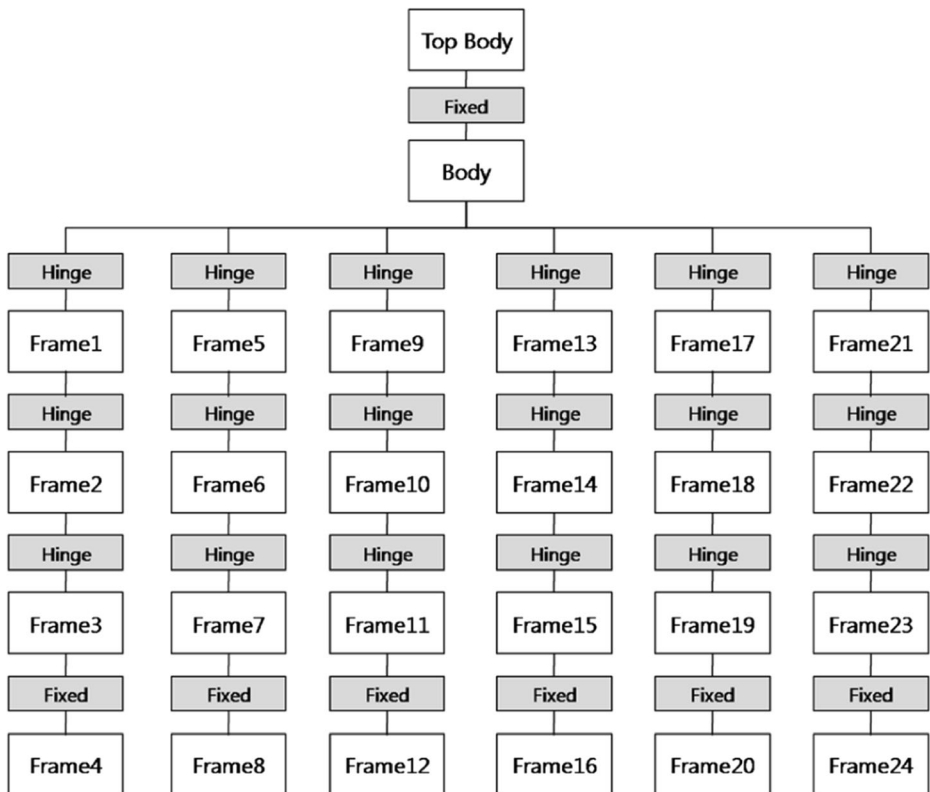
#### 4.2.2 Hardware component

The robot model is comprised of several hardware components connected to each other. The VirRobot has four types of parts: *Frame*, *Actuator*, *Body*, and *Sensor*. The *Component* in the robot meta-model defines each part.

Figure 8 represents the hardware components using 3D graphics along with their corresponding XML definitions. Each XML definition is composed of four elements: *type*, *id*, *name* and *graphic*. The *type* indicates the component's type as defined by the *ComponentType*, and the *id* is a unique identifier of the component itself. The *name* refers to the name of the

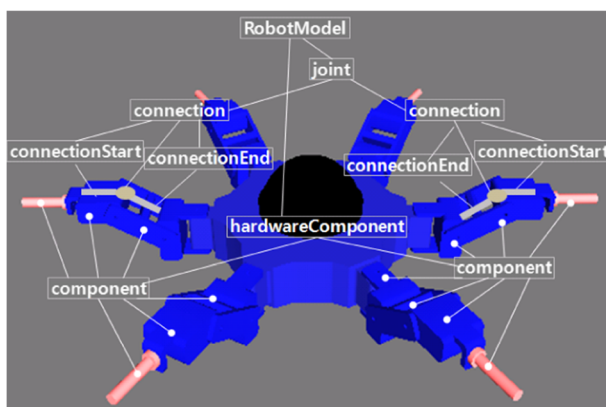


**Fig. 5** The six legged robot (made by KMC) and its VirRobot prototyping model










**Fig. 6** The structure of the robot model

component, and the *graphic* describes the file which stores the 3D graphic data. For example, in the robot model featured in Fig. 8, “*Leg4*” consists of two components: a “*Frame*” and an “*Actuator*” type. The Actuator component has no *graphic* element, but it does have a *refActionID*. For this reason, it is not displayed on the screen. This *refActionID* is connected to an API, and effectively defines the actions of the robot model.



**Fig. 7** Robot model mapped with the robot meta-model

Name	Parts	XML	Name	Parts	XML
Top Body		<pre>&lt;component type="Body" id="44" name="TopBody" graphic="body2.x" /&gt;</pre>	Leg2		<pre>&lt;component type="Frame" id="2" name="Leg2" graphic="leg2.x" /&gt; &lt;component type="Actuator" id="5" name="M1" refActionID="1" /&gt;</pre>
Body		<pre>&lt;component type="Body" id="43" name="Body" graphic="body1.x" refActionID="2" /&gt; &lt;component type="Fixed" id="45" name="TopBody.body" /&gt;</pre>	Leg1		<pre>&lt;component type="Frame" id="1" name="Leg1" graphic="leg1.x" /&gt; &lt;component type="Fixed" id="46" name="Leg2.Leg1" /&gt;</pre>
Leg4		<pre>&lt;component type="Frame" id="4" name="Leg4" graphic="leg4.x" /&gt; &lt;component type="Actuator" id="7" name="M3" refActionID="1" /&gt;</pre>	Sensor		<pre>&lt;component type="Sensor" id="45" name="U1" refActionID="3" /&gt; &lt;component type="Frame" id="46" name="Sensor1" graphic="sensor.x" /&gt;</pre>
Leg3		<pre>&lt;component type="Frame" id="3" name="Leg3" graphic="leg3.x" /&gt; &lt;component type="Actuator" id="6" name="M2" refActionID="1" /&gt;</pre>			

**Fig. 8** Hardware components and XML definitions

#### 4.2.3 Joint

A *Joint* connects two hardware components together, but is not displayed in 3D display. The *refComponentID* in the robot meta-model indicates components connected to one another. In accordance with the meta-model defined in Fig. 4, there are 4 types of *Joints*: *Fixed*, *Hinge*, *Ball*, and *Slider* because of using Open Dynamics Engine (ODE) [24].



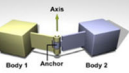
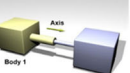
The *Fixed Joint* connects two components firmly together. The *Ball Joint* provides 360-degree component rotation using two different motors. The *Hinge Joint* allows 0–180-degree horizontal movement and commonly uses Servo motors. The *Slider Joint* can allow for vertical movement as similar to a slider. Figure 9 describes these four types of *Joints* and their corresponding XML definitions.

#### 4.2.4 MotionData





Motion data refers to an ordered sequence of the motor's value as a determinant of the robot's actual movement. Each XML definition is composed of two elements, *id* and *value*, and every robot action requires a sequence of motions. For example, the “*forwarding action*” requires four sequential motions as described in Fig. 10.

#### 4.2.5 Robot API for controlling motors

Hardware components such as motors have APIs to control them. Similarly, a VirRobot model allows both existing and newly created APIs to be used. The Robot API *refAction* is defined as a set consisting of an attribute *attr* and a method *method* in Fig. 11. The *attr* is composed of a *name* and an API *type*. The method consists of a *name* and *returnType* for the method as well as a *name* and *type* for its parameter. At next transformation phase, the API “*Motor*” will be transformed to a ‘motor’ class at the next transformation phase in Fig. 21.

Name	Joint	XML	Name	Joint	XML
Fixed		<pre>&lt;connection id="2" type="Fixed" name="C1" refComponentID="5"&gt; &lt;connectionStart name="C1.e1" refComponentID="1" /&gt; &lt;connectionEnd name="C1.e2" refComponentID="2" /&gt; &lt;/connection&gt;</pre>	Ball		<pre>&lt;connection id="2" type="Ball" name="C2" refComponentID="5"&gt; &lt;connectionStart name="C2.e1" refComponentID="1" /&gt; &lt;connectionEnd name="C2.e2" refComponentID="2" /&gt; &lt;/connection&gt;</pre>
Hinge		<pre>&lt;connection id="2" type="Hinge" name="C1" refComponentID="5"&gt; &lt;connectionStart name="C1.e1" refComponentID="1" /&gt; &lt;connectionEnd name="C1.e2" refComponentID="2" /&gt; &lt;/connection&gt;</pre>	Slider		<pre>&lt;connection id="2" type="Slider" name="C3" refComponentID="5"&gt; &lt;connectionStart name="C3.e1" refComponentID="1" /&gt; &lt;connectionEnd name="C3.e2" refComponentID="2" /&gt; &lt;/connection&gt;</pre>

**Fig. 9** Joints and their XML definitions

Name	Motion	XML	Name	Motion	XML
Forward Step 1		<code>&lt;data id="1" value="74 86 157 116 56 123 71 83 155 76 125 65 121 94 24 71 125 40 3 2" /&gt;</code>	Forward Step 3		<code>&lt;data id="3" value="114 56 142 66 86 153 106 53 115 111 95 25 71 124 56 111 95 25 3 2" /&gt;</code>
Forward Step 2		<code>&lt;data id="2" value="74 56 127 116 86 153 71 53 125 76 95 25 121 124 56 71 95 25 3 1" /&gt;</code>	Forward Step 4		<code>&lt;data id="4" value="114 86 157 66 56 123 106 83 155 111 125 55 71 94 24 111 125 53 1" /&gt;</code>

**Fig. 10** A sequence of robot motions for a forwarding action

### 4.3 Meta-modeling for the UML class model

The meta-model of UML 2.2 [25] has different compliance levels ranging in value from 1 to 3. A full scale meta-model represents a compliance level 3. It contains 14 language units and 39 packages as presented in Table 1. Among them, the language unit is the class diagram. Also, the package relevant to the case study are the “Classes”, “Classes::Kernel”, “Classes::Dependencies”, “Classes::Interfaces”, “Classes::AssociationClasses”, and “Classes::PowerTypes”. There are included 61 classes in these five packages.

Our proposed automatic transformation technique does not require the use of all 61 classes defined in the UML 2.2 meta-model, but rather need only 29 classes. Figure 12 depicts the class diagram reorganized with the appropriate classes. Although the structure of the UML 2.2 meta-model remains the same abstract classes, irrelevant classes are removed. Therefore, in the transformation phase a class diagram can be produced according to the reorganized UML meta-model.

## 5 Transformation from robot model into design model

In the transformation phase, a VirRobot model is transformed into a UML class diagram using the automatic transformation technique developed with ATL. The ATL transformation is composed of three sets or categories of rules: *Creation*, *Modification* and *Move*. The *Creation* rule creates the software architecture and structure of the multi-jointed robots. The VirRobot prototyping model then adopts a specific predefined software architecture, after which the rule requires no further input model or data. The *Modification* rule modifies the data values used in appropriate forms. The *Move* rule moves the modified data into the UML design model based on the robot model. Table 2 provides a summary of these categorized transformation rules. If need the whole rules written in ATRAS Transformation Language (ATL), you may visit our website [26].

```

<refAction id="1" name="Motor">
  <attr name="id" type="Integer"/>
  <attr name="graphic" type="String"/>
  <attr name="name" type="String"/>
  <method name = "SetDegree" returnType="void" >
    <parameter name="degree" type="Integer" />
  </method>
  <method name = "Stop" returnType="void" />
  <method name = "Start" returnType="void" />
</refAction>

```

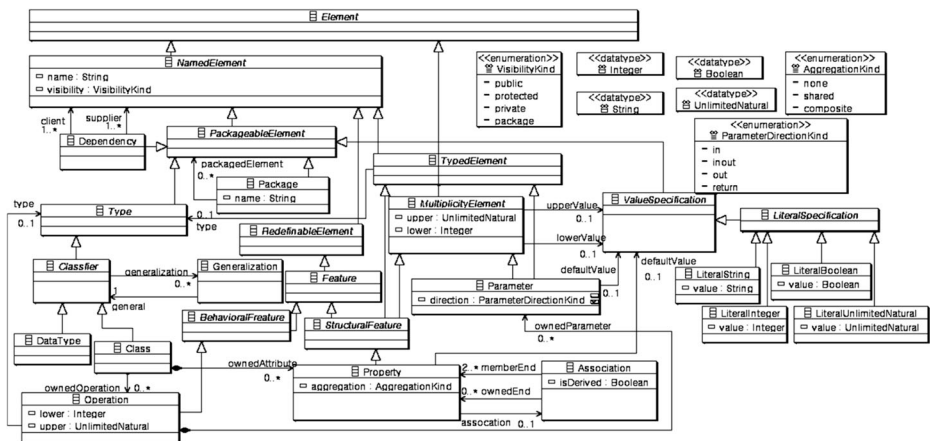
**Fig. 11** An XML definition of the Robot API “Motor”

**Table 1** The Language units and packages for UML 2.2 meta-model

Language Unit	Meta-Model Packages	
Actions	Actions::BasicActions	Actions::IntermediateActions
	Actions::StructuredActions	Actions::CompleteActions
Activities	Activities::FundamentalActivities	Activities::CompleteActivities
	Activities::BasicActivities	Activities::CompleteStructuredActivities
	Activities::IntermediateActivities	Activities::ExtraStructuredActivities
	Activities::StructuredActivities	
Classes	Classes::Kernel	Classes::AssociationClasses
	Classes::Dependencies	Classes::PowerTypes
	Classes::Interfaces	
Components	Components::BasicComponents	Components::PackagingComponents
Deployments	Deployments::Artifacts	Deployments::ComponentDeployments
	Deployments::Nodes	
General Behavior	CommonBehaviors::BasicBehaviors	CommonBehaviors::SimpleTime
	CommonBehaviors::Communications	
Structures	CompositeStructure::InternalStructures	CompositeStructures::StructuredClasses
	CompositeStructures::InvocationActions	CompositeStructures::Collaborations
	CompositeStructures::Ports	CompositeStructures::StructuredActivities
Interactions	Interactions::Fragments	Interactions::BasicInteractions
State Machines	StateMachines::BehaviorStateMachines	StateMachines::ProtocolStateMachines
UseCases	UseCases	
Information Flows	AuxilliaryConstructs::InformationFlows	
Models	AuxilliaryConstructs::Models	
Templates	AuxilliaryConstructs::Templates	
Profiles	AuxilliaryConstructs::Profiles	

## 5.1 (ATL\_TR1) creating a DataType

The first step in creating a class diagram is to generate a list of data type. Examples of *DataType* include *String*, *Integer*, and *Boolean*. These types are generated by the current version of the translator developed by our research team. Since this transformation rule does not need robot model as input, this step can be performed at once. Figure 13 describes the transformation process of ATL\_TR1 in detail.

**Fig. 12** A meta-model of the UML 2.2 class diagram

**Table 2** Categorized Transformation Rules

Category	Transformation Rule	
Creation	ATL_TR1	Creating DataType
	ATL_TR2	Creating Data Class
	ATL_TR3	Creating Motion Class
	ATL_TR4	Creating Dependency between Data and Motion
Modification	ATL_TR5	Creating Subclass of Motion Class
Move	ATL_TR6	Creating Class using RefAction
	ATL_TR7	Creating Association using Mapping
	ATL_TR8	Creating Association using Joint

## 5.2 (ATL\_TR2) creating a data class

It contains motion data composed of an identifier and motion values. The ATL\_TR2 creates a *Data* class and two member variables, *id* and *value*. It needs no input data, and only to be performed at one time. Figure 14 describes the transformation process of ATL\_TR2 in detail.

## 5.3 (ATL\_TR3) creating a motion class

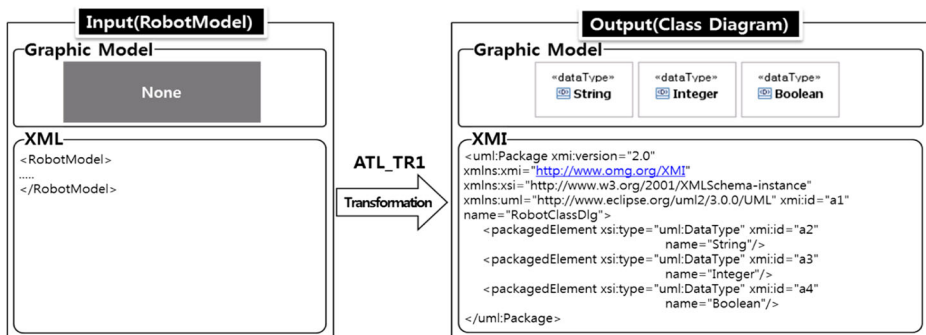
ATL\_TR3 creates a *Motion* class, storing one motion by combining several *Data* classes. It may have subclasses corresponding to each motion. It is also executed only once. Figure 15 describes the transformation process of ATL\_TR3 in detail.

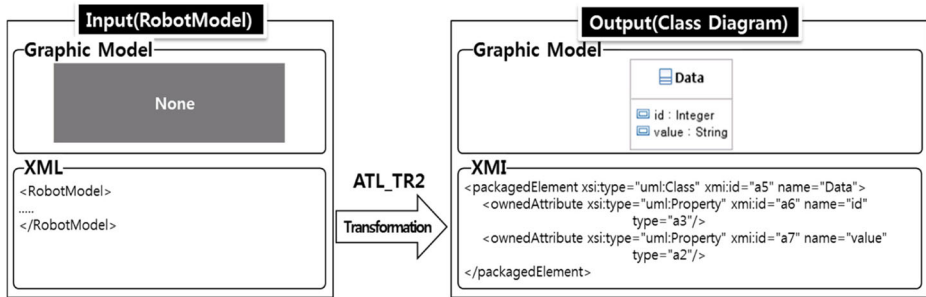
## 5.4 (ATL\_TR4) creating a dependency between a given data and motion

The *Motion* class and *Data* class are dependent on each other since the former contains the latter as a method. The ATL\_TR4 creates a dependency between them. This rule is executed only once regardless of the input from the robot model. Figure 16 describes the transformation process of ATL\_TR4 in detail.

## 5.5 (ATL\_TR5) creating a subclass of a motion class

ATL\_TR5 reads the robot's motion data and creates subclasses for the robot's motions. The name of a given robot motion corresponds to the name of the subclass translated for it. In the

**Fig. 13** Transformation Rule 1 (ATL\_TR1)



**Fig. 14** Transformation Rule 2 (ATL\_TR2)

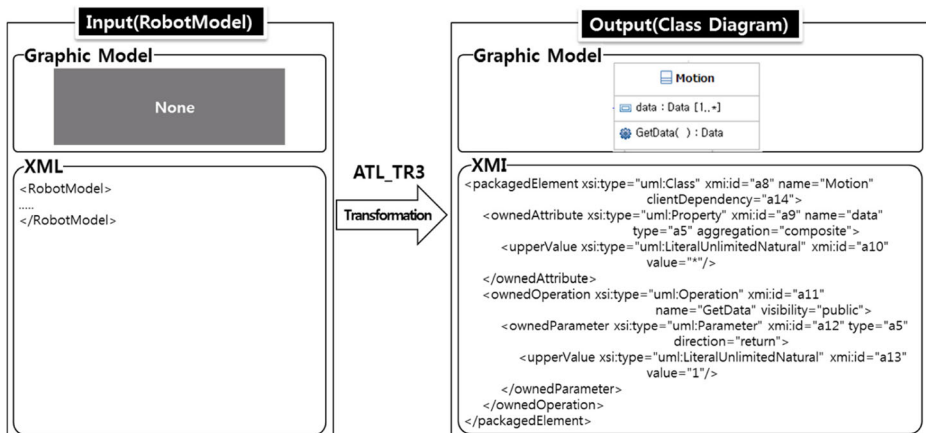
example shown in Fig. 18, the motion “*Forward*” is also the name of the transformed subclass. All *data* elements of the robot model are then transformed into *Data* objects. Each *Data* object is created using the “*new*” command. The created “*Forward*” class has a generalization relationship with its parent class *Motion*. Details of the transformation process for ATL-TR5 are described in Fig. 17.

## 5.6 (ATL\_TR6) creating a class using RefAction

*RefAction* in the robot model describes action APIs which the robot can perform. In the meta-model, each *RefAction* is transformed into a class with the same name. Thus, the *attr*, *method*, and *parameter* elements in the *RefAction* are transformed into member variable, member function, and member function parameters respectively. Figure 18 describes this transformation in detail.

## 5.7 (ATL\_TR7) creating an association using mapping

Transformation rule 7, ATL\_TR7 associates the *System* class with the *Motion* classes by using the *Mapping* function of the robot model. Mapping of the input model essentially connects the system with its motions. This means that motion data can be imported by the system, thereby enabling the multi-jointed robot to move. This connection generates a correlation between the



**Fig. 15** Transformation Rule 3 (ATL\_TR3)

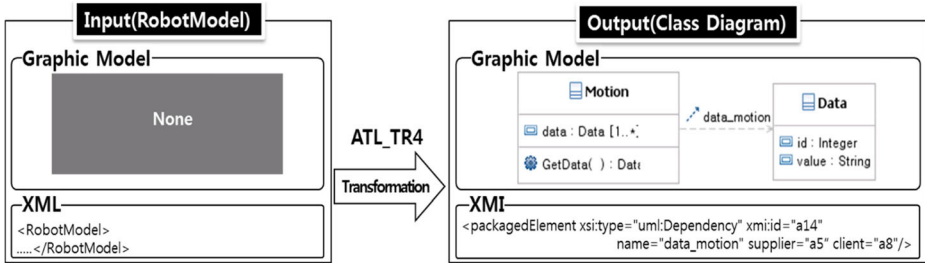


Fig. 16 Transformation Rule 4 (ATL\_TR4)

system class and the motion class based on the model conversion process. Figure 19 described the details of this transformation step.

### 5.8 (ATL\_TR8) creating an association using mapping

ATL\_TR8 represents a more complicated transformation procedure since it incorporates in data from joints. This in turn contains several *Connections* that refer to various hardware components.

Joints consist of multiple connections, and each connection links different hardware components together. It is worth mentioning that occasionally there are fixed components and hardware components that do not function properly in a joint (i.e., components expressing graphics only). Subsequently, they must be separated from the rest. Therefore, prior to any model conversion, special conditions must be checked and adjustments must be made such as the simplification of complex areas using an ATL helper. *Joints* are structures connected in pairs containing *Connection* - *ConnectionStart* - *ConnectionEnd*. This is similar to the structure of *Hardware Component* - *Hardware Component* - *Hardware Component*. As this form generates similar correlations between classes, a one-to-one conversion of the joints becomes possible in a class diagram. The conditions to satisfy this include the satisfaction of Condition 1 or Condition 2, which is presented in Table 3. This means that the connection's *refComponetID* must not equal -1 while

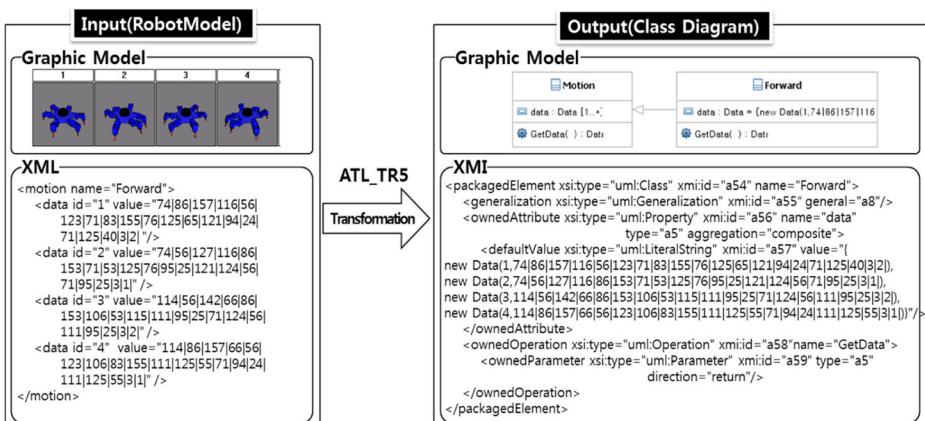
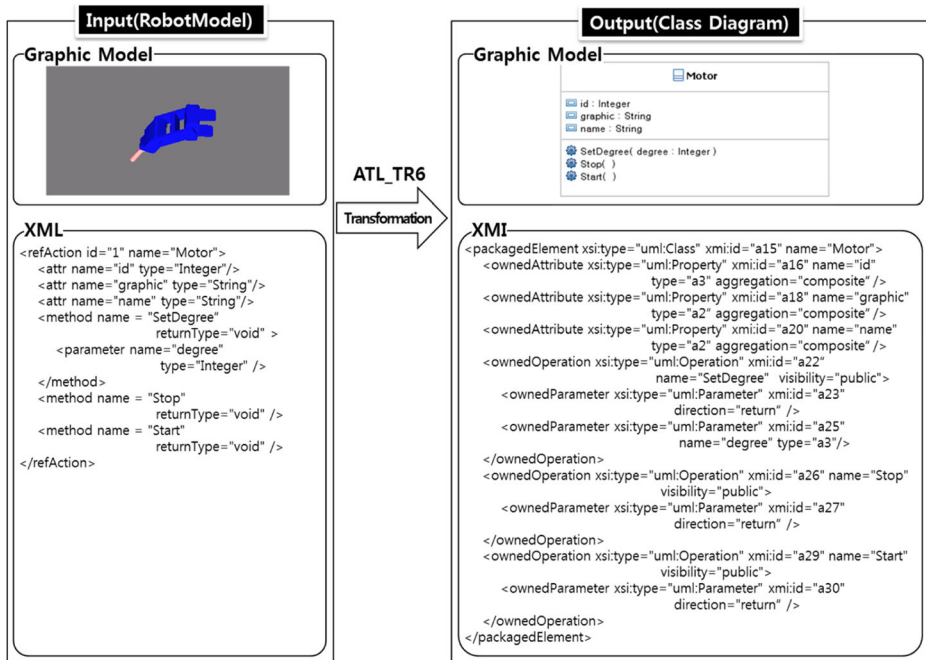


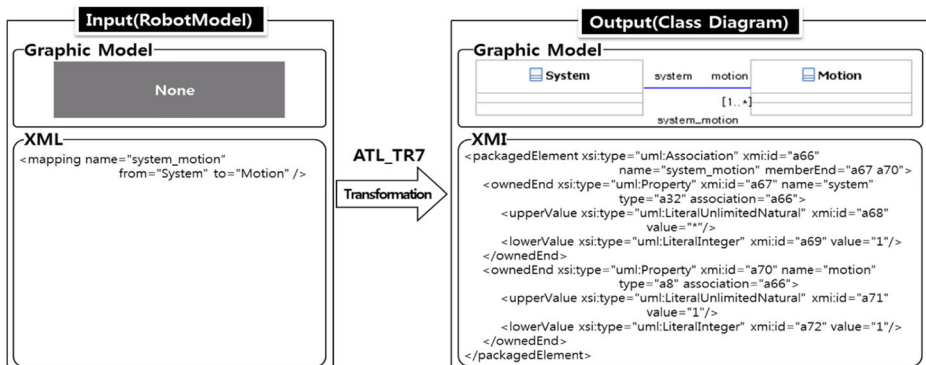
Fig. 17 Transformation Rule 5 (ATL\_TR5)



**Fig. 18** Transformation Rule 6 (ATL\_TR6)

*ConnectionStart* or *ConnectionEnd* has *RefAction*. Table 3 shoes conditions for the Class conversion of a Joint.

A *refComponentID* value of  $-1$  is used when simply connecting hardware components together as a fixed form in the model. For the creation of a class diagram, this is unnecessary. Also, the absence of *refActionID* in *ConnectionStart* and *ConnectionEnd* is not related to the class diagram itself, since it is a graphic-type hardware component. When one or more of the two aforementioned conditions are met, a correlation is generated in the class diagram pertaining to both conditions. Figure 20 provides a description of this conversion method in more detail.



**Fig. 19** Transformation Rule 7 (ATL\_TR7)

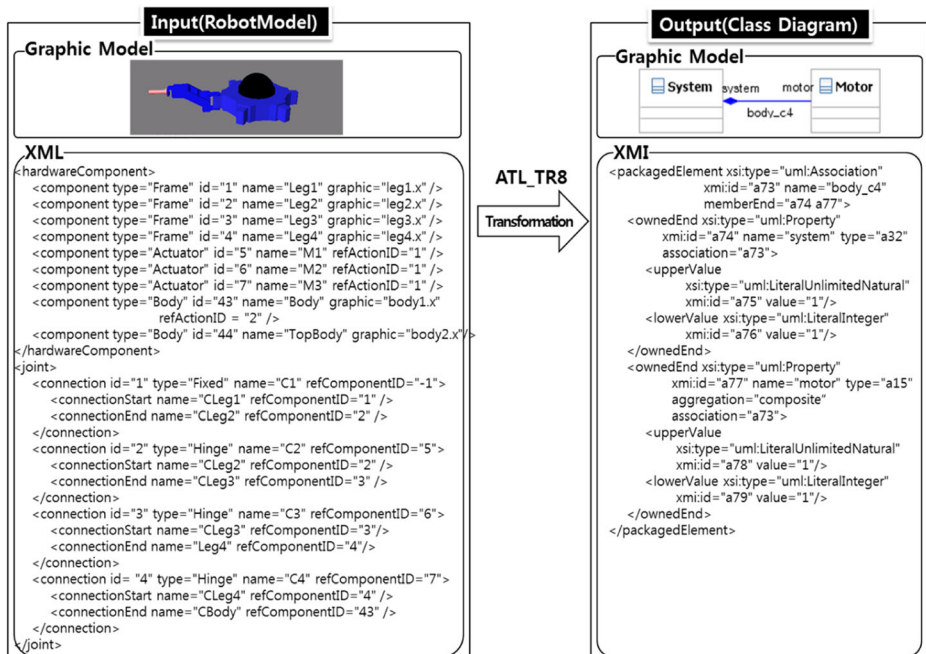
**Table 3** Conditions for the Class Conversion of a Joint

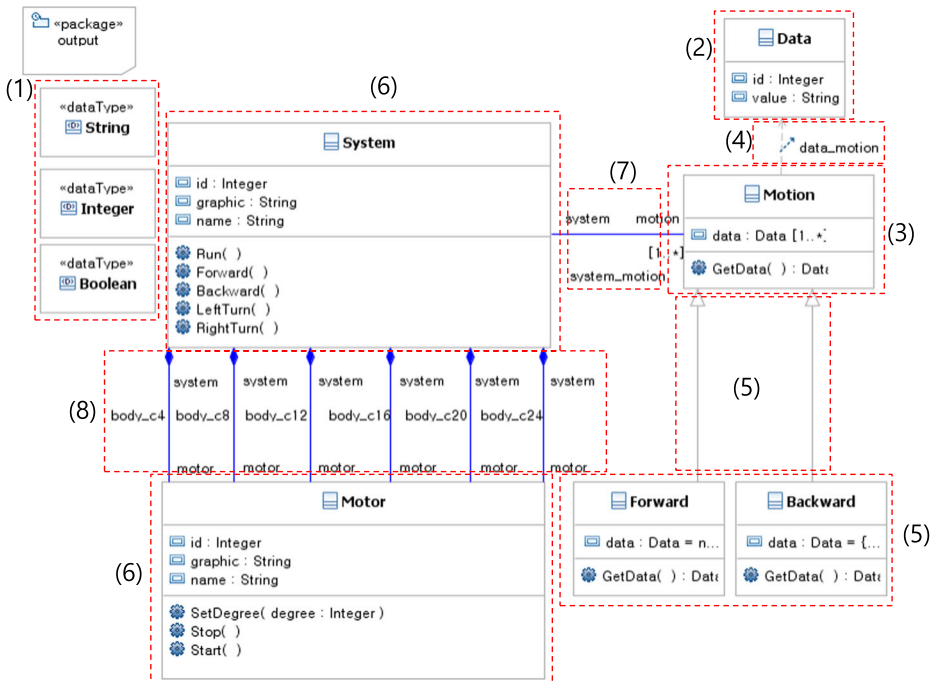
Joint	Condition 1	Condition 2
Connection	Not −1	Not −1
ConnectionStart	isRefAction	—
ConnectionEnd	—	isRefAction

## 5.9 Transforming design model with the ATL rules

The ATL transformation rules (ATL\_TR1 ~ ATL\_TR8) transform the VirRobot model depicted in Section 4.2 into a UML class diagram described in Fig. 21. For these conditions, the robot model previously mentios has six legs, one head, and one body. The transformed class diagram consists of six classes and ten associations. The *System* is the class which controls the robot. The *Motor* class controls the motors used in the multi-jointed robot. The *Motion* class defines the action data which consists of *Forward* and *Backward* subclasses. These subclasses in turn define motions of going forward and backward, respectively.

A class diagram can be generated based on the model conversion rules presented. In Fig. 21, especially, (1) shows the *DataType* which covers String, Integer, and Boolean. These are the basic variable types used in the models. (2) indicates the Data class that enables the recoding of one motion value. (3) denotes the Motion class that manages motion data necessary to conduct one movement. The Data class is used to insert the Motion data into appropriate attributes. (4) the Data class uses motion data inside a

**Fig. 20** Transformation Rule 8 (ATL\_TR8)



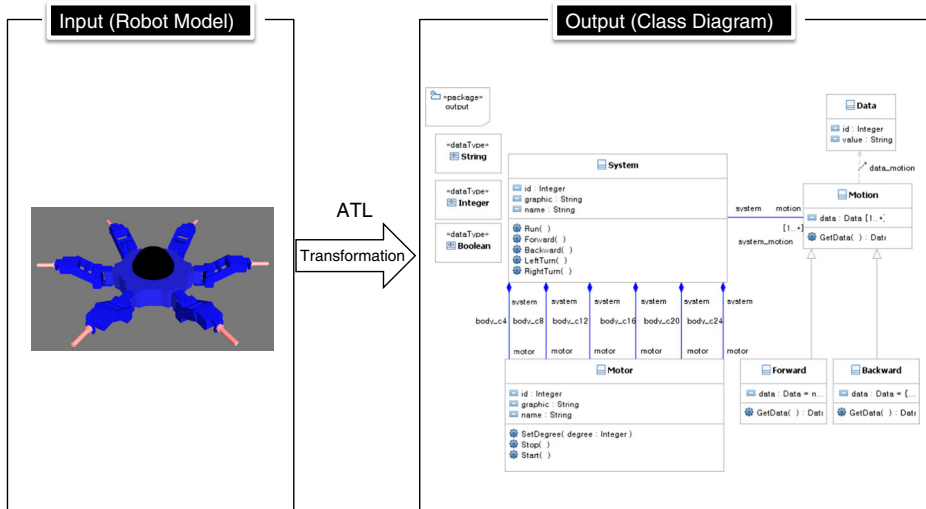
**Fig. 21** A class diagram transformed from the robot model

given function which results in a dependent relationship being created between the two classes. (5) is where the class to express the behaviors is generated in the fed model, thereby sub-classing the motion class. The current class has only Forward and Backward options. (6) involves system and motor class using the Robot API created in the fed model. Depending on the condition of the Robot API, different classes may be generated. (7) shows that the system class uses motions to conduct behaviors which correlate this data between and among each other. (8) further provides the correlation between the system and motor class generated by the joints themselves.

If we use a robot model which is more complicated than the above, the class diagram transformed may be more complicated too. We can also associate existing library APIs with the class diagram and generate source program code.

## 6 Experimental validation

To demonstrate the effectiveness and validity of the proposed transformation technique by examining multi-jointed robot models from the VirRobot prototyping, we evaluate about that two multi-jointed models are six-legged robots either with or without sensors. First, our experts manually develop class diagrams for each of these models with the UML tool, RUT. Second, we also automatically transform the robot models into UML models respectively. To validate the generated XMI code based on our suggested approach, we compare them with manually developed XMI codes.



(a) Robot Model

(b) UML class diagram

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<uml:Package xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UMML" xmi:id="a1" name="RobotClassDlg">
  <packagedElement xsi:type="uml:DataType" xmi:id="a2" name="String">
  ...
  <packagedElement xsi:type="uml:Class" xmi:id="a5" name="Data">
    <ownedAttribute xsi:type="uml:Property" xmi:id="a6" name="id" type="a3"/>
    <ownedAttribute xsi:type="uml:Property" xmi:id="a7" name="value" type="a2"/>
  </packagedElement>
  <packagedElement xsi:type="uml:Class" xmi:id="a8" name="Motion" clientDependency="a14">
    <ownedAttribute xsi:type="uml:Property" xmi:id="a9" name="data" type="a5" aggregation="composite">
      <upperValue xsi:type="uml:LiteralUnlimitedNatural" xmi:id="a10" value="*" />
    </ownedAttribute>
    <ownedOperation xsi:type="uml:Operation" xmi:id="a11" name="GetData" visibility="public">
      <ownedParameter xsi:type="uml:Parameter" xmi:id="a12" type="a5" direction="return">
        <upperValue xsi:type="uml:LiteralUnlimitedNatural" xmi:id="a13" value="1" />
      </ownedParameter>
    </ownedOperation>
  </packagedElement>
  <packagedElement xsi:type="uml:Dependency" xmi:id="a14" name="data_motion" supplier="a5" client="a8">
  ...
    <ownedParameter xsi:type="uml:Parameter" xmi:id="a25" name="degree" type="a3"/>
  ...
  <packagedElement xsi:type="uml:Class" xmi:id="a60" name="Backward">
    <generalization xsi:type="uml:Generalization" xmi:id="a61" general="a8"/>
  ...
  </packagedElement>
  <packagedElement xsi:type="uml:Association" xmi:id="a66" name="system_motion" memberEnd="a67 a70">
    <ownedEnd xsi:type="uml:Property" xmi:id="a67" name="system" type="a32" association="a66">
      <upperValue xsi:type="uml:LiteralUnlimitedNatural" xmi:id="a68" value="*" />
      <lowerValue xsi:type="uml:LiteralInteger" xmi:id="a69" value="1" />
    </ownedEnd>
    <ownedEnd xsi:type="uml:Property" xmi:id="a70" name="motion" type="a8" association="a66">
      <upperValue xsi:type="uml:LiteralUnlimitedNatural" xmi:id="a71" value="1" />
      <lowerValue xsi:type="uml:LiteralInteger" xmi:id="a72" value="1" />
    </ownedEnd>
  </packagedElement>
</uml:Package>

```

(c) The XMI's result of UML class diagram

**Fig. 22** An overview of the 6-legged robot's transformation

**Table 4** Comparison of both automatically transformed XMI code and manually developed one of UML class diagram for six-legged robot (T: Type, A: Automatic Generation, M: Manual Development)

Components of meta class	T	The XMI code of UML class diagram
DataType	A	<packagedElement xsi:type = “uml:DataType” xmi:id = “a2” name = “String”/>
	M	<packagedElement xmi:type = “uml:DataType” xmi:id = “_Iac0WotfEd6OJ99i0DhBUw” name = “String” />
Class	A	<packagedElement xsi:type = “uml:Class” xmi:id = “a5” name = “Data” > </packagedElement>
	M	<packagedElement xmi:type = “uml:Class” xmi:id = “_Iac0SetfEd6OJ99i0DhBUw” name = “Data” > </packageElement>
Attribute	A	<ownedAttribute xsi:type = “uml:Property” xmi:id = “a9” name = “data” type = “a5” aggregation = “composite”> <upperValue xsi:type = “uml:LiteralUnlimitedNatural” xmi:id = “a10” value = “*”/> </ownedAttribute>
	M	<ownedAttribute xmi:id = “_Iac0TetfEd6OJ99i0DhBUw” name = “data” type = “_Iac0SetfEd6OJ99i0DhBUw” aggregation = “composite”> <upperValue xmi:type = “uml:LiteralUnlimitedNatural” xmi:id = “_Iac0TutEd6OJ99i0DhBUw” value = “*”/> </ownedAttribute>
Operation	A	<ownedOperation xsi:type = “uml:Operation” xmi:id = “a11” name = “GetData” visibility = “public”> <ownedParameter xsi:type = “uml:Parameter” xmi:id = “a12” type = “a5” direction = “return”> <upperValue xsi:type = “uml:LiteralUnlimitedNatural” xmi:id = “a13” value = “1”/> </ownedParameter></ownedOperation>
	M	<ownedOperation xmi:id = “_Iac0T-tfEd6OJ99i0DhBUw” name = “GetData”> <ownedParameter xmi:id = “_Iac0UotfEd6OJ99i0DhBUw” type = “_Iac0SetfEd6OJ99i0DhBUw” direction = “return”> <upperValue xmi:type = “uml:LiteralUnlimitedNatural” xmi:id = “_Iac0UetfEd6OJ99i0DhBUw” value = “1”/> </ownedParameter></ownedOperation>
Parameter	A	<ownedParameter xsi:type = “uml:Parameter” xmi:id = “a25” name = “degree” type = “a3”/>
	M	<ownedParameter xmi:id = “_Iac0ROtfEd6OJ99i0DhBUw” name = “degree” />
Association	A	<packagedElement xsi:type = “uml:Association” xmi:id = “a66” name = “system_motion” memberEnd = “a67 a70”> <ownedEnd xsi:type = “uml:Property” xmi:id = “a67” name = “system” type = “a32” association = “a66”> <upperValue xsi:type = “uml:LiteralUnlimitedNatural” xmi:id = “a68” value = “*”/> <lowerValue xsi:type = “uml:LiteralInteger” xmi:id = “a69” value = “1”/></ownedEnd> <ownedEnd xsi:type = “uml:Property” xmi:id = “a70” name = “motion” type = “a8” association = “a66”> <upperValue xsi:type = “uml:LiteralUnlimitedNatural” xmi:id = “a71” value = “1”/> <lowerValue xsi:type = “uml:LiteralInteger” xmi:id = “a72” value = “1”/> </ownedEnd></packagedElement>
	M	<packagedElement xmi:type = “uml:Association” xmi:id = “_Iac0Y-tfEd6OJ99i0DhBUw” name = “system_motion” memberEnd = “_Iac0ZotfEd6OJ99i0DhBUw _Iac0Z-tfEd6OJ99i0DhBUw”> <ownedEnd xmi:id = “_Iac0ZotfEd6OJ99i0DhBUw” name = “system” type = “_Iac0MotfEd6OJ99i0DhBUw” association = “_Iac0Y-tfEd6OJ99i0DhBUw”> <upperValue xmi:type = “uml:LiteralUnlimitedNatural” xmi:id = “_Iac0ZetfEd6OJ99i0DhBUw” value = “1”/> <lowerValue xmi:type = “uml:LiteralInteger” xmi:id = “_Iac0ZutEd6OJ99i0DhBUw” value = “1”/> </ownedEnd> <ownedEnd xmi:id = “_Iac0Z-tfEd6OJ99i0DhBUw” name = “motion” type = “_Iac0TOtfEd6OJ99i0DhBUw” association = “_Iac0Y-tfEd6OJ99i0DhBUw”> <upperValue xmi:type = “uml:LiteralUnlimitedNatural” xmi:id = “_Iac0aOtfEd6OJ99i0DhBUw” value = “*”/>

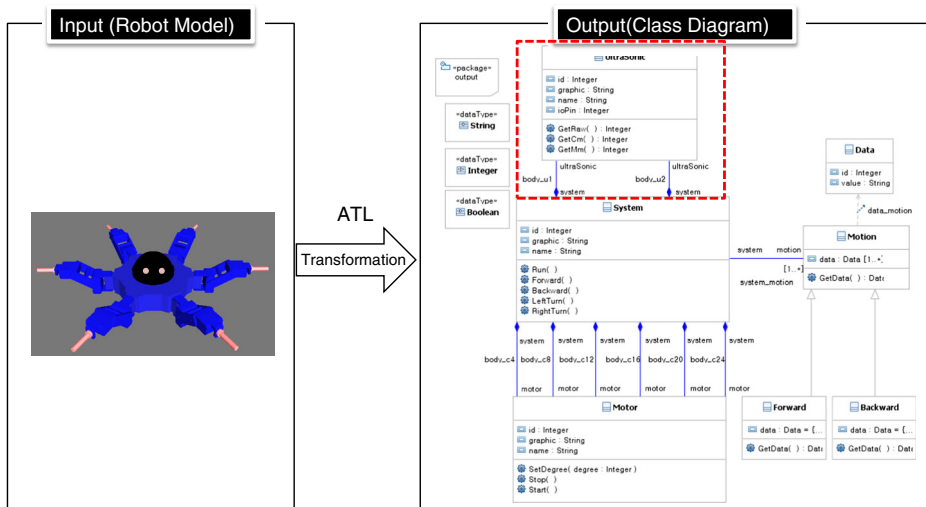
**Table 4** (continued)

Components of meta class	T	The XMI code of UML class diagram
		<pre> &lt;lowerValue xmi:type="uml:LiteralInteger" xmi:id="_Iac0aetfEd6OJ99i0DhBUw" value="1"/&gt; &lt;/ownedEnd&gt;&lt;/packagedElement&gt; </pre>
Dependency	A	<pre> &lt;packagedElement xsi:type="uml:Dependency" xmi:id="a14" name="data_motion" supplier="a5" client="a8"/&gt; </pre>
	M	<pre> &lt;packagedElement xmi:type="uml:Dependency" xmi:id="_Iac0YutfEd6OJ99i0DhBUw" name="data_motion" supplier="_Iac0SetfEd6OJ99i0DhBUw" client="_Iac0TOtfEd6OJ99i0DhBUw"/&gt; </pre>
Generalization	A	<pre> &lt;generalization xsi:type="uml:Generalization" xmi:id="a61" general="a8"/&gt; </pre>
	M	<pre> &lt;generalization xmi:id="_Iac0U-tfEd6OJ99i0DhBUw" general="_Iac0TOtfEd6OJ99i0DhBUw"/&gt; </pre>

## 6.1 A first case: The six legged robot

We show the transformation process for the six-legged multi-jointed robot model in Fig. 22. With input model(that is, Robot model), we can transform with ATL rules to create the *System*, *Motor*, *Motion*, *Data*, *Forward*, and *Backward* classes along with associations for each of six legs. Like sections 4 and 5, the *Motion*, *Data*, *Forward*, and *Backward* classes were responsible for processing the robot's motions. Figure 22 shows an overview of the six-legged robot's transformation.

In Table 4, we show almost similarity between our automatic code generation (A) and our manual code development(M) on all elements of meta class. That is, we can recognize to have no differences between XMI code of the transformed class diagrams and one developed manually using a UML tool. It further shows that both XMI codes are the same except generating the unique ID.

**Fig. 23** The six-legged robot's transformation with two sensors

**Table 5** Comparison of both automatically transformed XMI code and manually developed one of UML class diagram for six-legged robot with two sensors (C: Components of meta class, T: Type, A: Automatic Generation, M: Manual Development)

C	T	The XMI code of UML class diagram
Class	A	<packagedElement xsi:type = “uml:Class” xmi:id = “a54” name = “UltraSonic”> </packagedElement>
	M	<packagedElement xmi:type = “uml:Class” xmi:id = “_0hfWQQX4Ed-KYvZw-Au8DQ” name = “UltraSonic”> </packagedElement>
Association	A	<packagedElement xsi:type = “uml:Association” xmi:id = “a133” name = “body_ul” memberEnd = “a134 a137”> <ownedEnd xsi:type = “uml:Property” xmi:id = “a134” name = “system” type = “a32” association = “a133”> <upperValue xsi:type = “uml:LiteralUnlimitedNatural” xmi:id = “a135” value = “1”/> <lowerValue xsi:type = “uml:LiteralInteger” xmi:id = “a136” value = “1”/> </ownedEnd> <ownedEnd xsi:type = “uml:Property” xmi:id = “a137” name = “ultrasonic” type = “a54” aggregation = “composite” association = “a133”> <upperValue xsi:type = “uml:LiteralUnlimitedNatural” xmi:id = “a138” value = “1”/> <lowerValue xsi:type = “uml:LiteralInteger” xmi:id = “a139” value = “1”/> </ownedEnd> </packagedElement>
	M	<packagedElement xmi:type = “uml:Association” xmi:id = “_0hfWfgX4Ed-KYvZw-Au8DQ” name = “body_ul” memberEnd = “_0hfWfwX4Ed-KYvZw-Au8DQ _0hfWggX4Ed-KYvZw-Au8DQ”> <ownedEnd xmi:id = “_0hfWfwX4Ed-KYvZw-Au8DQ” name = “system” type = “_0hfWEAX4Ed-KYvZw-Au8DQ” association = “_0hfWfgX4Ed-KYvZw-Au8DQ”> <upperValue xmi:type = “uml:LiteralUnlimitedNatural” xmi:id = “_0hfWgAX4Ed-KYvZw-Au8DQ” value = “1”/> <lowerValue xmi:type = “uml:LiteralInteger” xmi:id = “_0hfWgQX4Ed-KYvZw-Au8DQ” value = “1”/> </ownedEnd> <ownedEnd xmi:id = “_0hfWggX4Ed-KYvZw-Au8DQ” name = “ultraSonic” type = “_0hfWQQX4Ed-KYvZw-Au8DQ” aggregation = “composite” association = “_0hfWfgX4Ed-KYvZw-Au8DQ”> <upperValue xmi:type = “uml:LiteralUnlimitedNatural” xmi:id = “_0hfWgwX4Ed-KYvZw-Au8DQ” value = “1”/> <lowerValue xmi:type = “uml:LiteralInteger” xmi:id = “_0hfWhAX4Ed-KYvZw-Au8DQ” value = “1”/> </ownedEnd> </packagedElement>

## 6.2 A second case: Six-legged robot with two sensors

For more robot model’s case, we add two sensors to six-legged robot as depicted in Fig. 23. The whole generated classes of the second case are created with the original System class in addition to generating an *UltraSonic* class and two associations. The added parts of Fig. 23 are shown within a dotted box. Table 5 shows a comparison between the *Ultrasonic* class and its associations. No differences between them are detected except for the mechanism generating the unique ID.

Therefore, we validate our proposed transformation technique in that we find no differences between the automatically transformed XMI codes with the manually developed one.

## 7 Conclusion

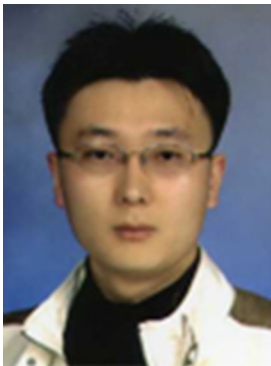
Most of robotic companies develop a control programming of multi-jointed robots, which spend too much time to manually adjust the moving functions of the robots. To solve this problem, we adapt the virtual prototyping (VP) to develop the control program of the robotic behaviors. For software engineers, in order for them to easily program this robot, we also adapt metamodel mechanism to convert UML models with virtual prototyping model. We developed the automatic model transformation from the virtual prototyping model to UML models, which then automatically develop coding with HIMEM. To prove our mechanism's efficiency, we implemented *Robot to UML Translator (RUT)* as our transformation rules with ATLAS transformational language. In the future, we intend to extend the automatic transformation technique using motion scripts. In order to make the simulation process more precise, we in turn should include both sequence and state diagrams, as well as class diagrams. In this regard, we expect to apply orbit tracing techniques that can enable softer movement. In addition, it will be added various sensors (e.g. temperature, GPS, and acceleration) and hardware components applicable to the model while the converter will be extended to further enable any model conversion.

**Acknowledgments** This work was supported by the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (NRF-2015H1C1A1035548)

## References

1. Raibert MH (1986) Legged Robots. *Commun ACM* 29(6):499–514
2. Kim WY, Son HS, Kim RYC, Carlson CR (2009) MDD based CASE Tool for Modeling Heterogeneous Multi Jointed Robots. In: *Proceedings of the 2009 World Congress on Computer Science and Information Engineering*, IEEE Computer Society, 31 March–2 April, LA, California USA, pp. 775–779
3. Kim DW, Son HS, Kim WY, Kim RYC (2008) Application of M&S (Modeling & Simulation) for the Autonomous Reconnaissance Ground Robot. In: *Proceedings of the 2008 Communication/Electron in Agency for Defense Development, Agency for Defense Development*, 23 October, Seoul, Korea, pp. 168–171
4. OMG (2006) Meta Object Facility (MOF) Core Specification Ver. 2.0. <http://www.omg.org/spec/MOF/2.0/PDF>. Accessed 2 April 2013
5. OMG (2014) Model Driven Architecture (MDA) Guide Rev:2.0 <http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf>. Accessed 6 May 2015
6. Son HS, Kim WY, Kim RYC (2008) Semi-Automatic Software Development based on MDD for Heterogeneous Multi-Joint Robots. In: *Proceedings of International Symposium on Control and Automation*, IEEE Computer Society, 13–15 December, Sanya, China, pp. 93–98
7. Grønmo R, Oldevik J (2005) An Empirical Study of the UML Model Transformation Tool (UMT). In: *Proc. First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA)*
8. Vojtisek D, Je'ze'quel J-M (2004) MTL and Umlaut NG: Engine and Framework for Model Transformation. *ERCIM News*. [http://www.ercim.org/publication/Ercim\\_News/enw58/vojtisek.html](http://www.ercim.org/publication/Ercim_News/enw58/vojtisek.html). Accessed 1 June 2015
9. OMG (2015) Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.2. <http://www.omg.org/spec/QVT/1.2/PDF>. Accessed 2 May 2016
10. Be'zivin J, Dupe' G, Jouault F, Pitette G, Rougui JE (2003) First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In: *Proc. Workshop on Generative Techniques in the Context of Model Driven Architecture*, pp. 1–18
11. Jouault F, Kurtev I (2005) Transforming Models with ATL. In: *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, Springer-Verlag Berlin, Heidelberg, 2–7 October, Montego Bay, Jamaica, pp. 128–138
12. Tseng MM, Jiao J, CJ S (1988) Virtual prototyping for customized product development. *Integr Manuf Syst* 9(6):334–343

13. Microsoft (2007) Microsoft Robotics Studio (MSRS) Introduction. <http://www.microsoft.com/robotics/2007>. Accessed 10 December 2012
14. Gassmann B, Scholl K-U, Berns K (2001) Locomotion of LAURON III in rough terrain. In: Proc. IEEE/ASME International Conference on Advanced Intelligent Mechatronics 2, pp. 959–964
15. Kerscher T, Albiez J, Berns K (2002) Joint control of the six-legged robot AirBug driven by fluidic muscles. In: Proceedings of third international workshop on robot motion and control, IEEE, 11 November, Bukowy Dworek, Poland, pp. 27–32
16. Grieco JC, Prieto M, Armada M, Gonzalez de Santos P (1998) A six-legged climbing robot for high payloads. In: Proc. IEEE international conference on control applications 1, pp. 446–450
17. Wettergreen D, Pangels H, Bares J (1995) Behavior-based gait execution for the Dante II walking robot. In: Proc. intelligent robots and systems 3, pp. 274–279
18. Kim JS, Son HS, Kim WY, Kim RYC (2008) A study on education software for controlling of multi-joint robot. J Korean Assoc Inf Educ 12(4):469–476
19. Jouault F, Kurtev I (2005) Transforming models with ATL. In: proc. satellite events at the MoDELS, LNCS 3844, pp. 128–138
20. OMG (2014) Object Constraint Language Specification Version 2.4. <http://www.omg.org/spec/OCL/2.4/PDF>. Accessed 2 May 2016
21. OMG (2014) XML Metadata Interchange (XMI) Specification Version 2.4.2, <http://www.omg.org/spec/XMI/2.4.2/PDF>. Accessed 2 May 2016
22. Kim WY, Son HS, Kim RYC (2008) Design automation for heterogeneous SUGVs with UML profile mechanism. J KIISE:Soft Appli 35(12):705–715
23. Kim WY, Kim RYC (2007) A study on modeling heterogeneous embedded S/W components based on model driven architecture with extended xUML. KIPS Trans: Part D 14-D(1):83–88
24. Smith R (2006) Open dynamics engine V0.5 user guide. <http://www.ode.org/ode-latest-userguide.pdf>. Accessed 1 June 2015
25. OMG (2009) OMG Unified Modeling Language (OMG UML) Superstructure Version 2.2. <http://www.omg.org/spec/UML/2.2/Superstructure/PDF>. Accessed 2 April 2013
26. ATL's full codes of robot transformation, [http://selab.hongik.ac.kr/~son/rob\\_to\\_class.atl](http://selab.hongik.ac.kr/~son/rob_to_class.atl)



**Hyun Seung Son** received the B.S. and M.S. degree in Software Engineering from Hongik University, Korea in 2009. He is currently a Ph.D. candidate in Hongik University. His research interests are in the areas of Automation Tool Development in Embedded Software, Real Time Operation System Development, Metamodel design, Model Transformation, and Model Verification & Validation Method.



**R. Young Chul Kim** received the B.S. degree in Computer Science from Hongik University, Korea in 1985, and the Ph.D. degree in Software Engineering from the department of Computer Science, Illinois Institute of Technology (IIT), USA in 2000. He is currently a professor in Hongik University. His research interests are in the areas of Test Maturity Model, Embedded Software Development Methodology, Model Based Testing, Metamodel, Business Process Model and User Behavior Analysis Methodology.