Object-Centric Programming: A New Modeling Paradigm for Robotic Applications

Andreas Angerer, Alwin Hoffmann, Frank Ortmeier, Michael Vistein and Wolfgang Reif
Institute for Software & Systems Engineering
University of Augsburg, Augsburg, Germany
{*angerer, alwin.hoffmann, ortmeier, vistein, reif*}*@informatik.uni-augsburg.de*

***Abstract*—During the last 15 years the way how software products are being developed has changed dramatically. Today, software is developed very efficiently in an industrialized manner. One of the cornerstones is that new development processes introduced a domain-centric view rather than a technology-centric view. As a result, big and complex software systems can be built very fast, reliable and according to customers' requirements. Unfortunately, these advances in software engineering have had little to no effect on the software development process for robotic applications. This paper explains how domain-centric design can be introduced in the domain of industrial robotics and which possible benefits it might yield.**

Index Terms*—*object-oriented design, software architectures, object-level robot programming

I. Introduction

Industrial robots nowadays are usually programmed with proprietary languages varying among manufacturers. Examples are the KUKA Robot Language [1], ABB's robot programming language RAPID [2], or KAREL developed by FANUC Robotics [3]. These languages are tailored to robot programming by using special commands for robot motion or controlling robot tools. However, these languages are rather old and are based on concepts of programming languages like Pascal, which date back to the 1970s. The common concept of those languages is programming on manipulator level, i.e. the programmer mainly writes control logic for the robot's end-effector. [4] gives an overview about manipulator-level programming languages and other robot programming paradigms like object-level programming, which is particularly interesting for this work.

Software development for robotic applications is often done as follows: For a given domain-specific problem, an expert in robot programming tries to understand the domain and then writes a program for the robot, such that the intended task is achieved. This can be done either directly in a robot programming language or for modern target hardware in high-level languages. However, manipulator-level programming still requires translating the specific domain problem into a solution on the level of the relevant manipulators. This translation process is time consuming and does not only require the knowledge of domain experts, but also a considerable expertise in robot programming. Moreover, there exists no process for systematically constructing such translations.

On the other hand, software engineering is developing more and more into an engineering discipline. Development processes (e.g. the Unified Process) greatly support development of all kinds of software. These processes provide methods for analyzing the application domain, capturing the results in a set of artifacts (graphical and textual documents) and finally transforming the results into a software design that fulfills the initial requirements. The concept of object-orientation is an important paradigm in today's software engineering, as it allows direct and systematic translation of real world concepts into software objects.

This work proposes an approach to introduce the advances in software engineering into the domain of robotics application development. One step towards this achievement is the use of object-level robot programming. It allows modeling applications on the level of the entities involved in the task rather than the manipulators as in traditional manipulator-level programming. Early object-level robot programming languages are RAPT [5], AL [6] and TORBOL [7]. By introducing object-oriented programming, this leads to an *object-centric* approach for modeling robotic applications. It can be seen as an enabler for the integration of robotic application development into standard software engineering processes. Consequently, software development becomes more systematic and efficient.

Section II presents the approach of object-centric modeling in detail and outlines its advantages over classical manipulator-level programming. A prototypical realization of a software architecture for developing object-centric applications is outlined in Section III. Concluding remarks and an outlook are given in Section IV.

II. Core concepts

Today's object-oriented software engineering approach is driven by creating an abstract model of the application domain and use it for the structure of the software that is constructed. Usually, the starting point is a description of the task the system has to achieve. This is often done by writing down use cases [8]. The next step focuses on identifying important concepts of the application domain, transforming

them into a hierarchy of classes, and adding their dependencies. In object-oriented analysis and design [9], the focus lies on the *task* that has to be be fulfilled and on the *domain* of the application rather than on the technical realization. As a consequence, software development has become more efficient and the resulting products are tailored to fulfill the requirements. Moreover, re-usability and maintainability are increased significantly. This leads to the question: "What benefits can be transferred into the domain of robotics?" In order to answer this questions, a traditional and an object-centric solution for a small example from the palletizing domain is compared. A prototypical realization for the object-centric solution will be shown afterwards in Section III.

A. Simple example task

Fig. 1 shows a simple example taken from a palletizing application. A common (sub-)task of such applications is to staple items onto each other. In the figure, the relevant objects are two boxes. This task is quite simple. Nevertheless, it is already complex enough to show the differences between traditional programming and object-centric solutions. Consider the following common variations in the task:

- *Dimensions or position:* If size or position of a workpiece vary, the same manipulator might still be able to handle it, but the exact movement has to be varied.
- *Shape and weight:* If shape and weight of workpieces vary, it might be necessary to use other or even multiple manipulators to move the item. Examples are lack of stiffness, heavy weight or the necessity to move/rotate the pallets in the example above.
- *Environment and obstacles:* A changing environment with new obstacles is a characteristic feature of many scenarios, e.g. in palettizing. Movement paths may have to be adapted to avoid collisions.

B. Manipulator-level solution to the example

A *classical* manipulator-level approach yields a solution like the pseudo code in the following listing:

```
Move Robot to point P1;
Close Gripper;
Move Robot to point P2;
Open Gripper;
```

Listing 1. Manipulator-level program for stacking boxes

The point *P1* represents the position/orientation where the robot has to be moved to for gripping the workpiece, and *P2* is where the robot has to be positioned so that the workpiece is located at its destination – on top of the other workpiece.

1) Dimensions or position: If size or position of a workpiece differs from the specification the program was originally developed for, the absolute positions in space of P1 and P2 have to be adapted. This can, in general, be handled by adequate parametrization of the robot program. An elegant way to deal with this case would be the introduction of

Fig. 1. Stacking boxes: an example task

data structures representing characteristic properties of a workpiece, like e.g. a point where it can be gripped. The common solution for this problem in robot programming languages is the definition of a special coordinate system for the description of points relative to workpieces. Certain cases, like changes of the position of the workpiece in workspace, can then be handled by adapting this coordinate system. This automatically causes the adaption of all points defined relative to the coordinate system.

2) Shape and weight: Changes in shape and weight of the workpiece that is to be transported generally cannot be covered by simple parametrization of a program on manipulator level. In case the workpiece's weight exceeds the load capacity of the commanded manipulator, further available manipulators could solve the problem. The same is true if a lack in the workpiece's stiffness requires it to be gripped at several sides for transportation. Imagine a scenario in which two robots palletize goods in the same work cell. Normally they should do the palletizing task in parallel and independently from each other to achieve a high throughput of workpieces. Only in situations where load sharing is required they should work together carrying one workpiece.

A manipulator-level program that can cope with this case completely differs from the *basic program* shown in Listing 1. The following listing shows in pseudo code how the program has to be extended for load sharing:

```
Move Robot1 to P1a;
Close Gripper1;
Move Robot2 to P1b;
Close Gripper2;
Sync Robot1 and Robot2;
Move Robot1 (and Robot2) to P2;
Open Gripper1;
Open Gripper2;
```

Listing 2. Manipulator-level program for stacking boxes with load sharing

P1a and P1b represent points where the two grippers should take the workpiece. For every additional robot, the program gets more complex. this complexity has to be considered initially, otherwise the program has to be rewritten when such workplaces come into play.

3) Environment and obstacles: Variations in the movement path almost always lead to changes in manipulator-level programs. As the movements are usually defined as a sequence of single motions between some defined points, moving the robot and its workpiece around obstacles requires the definition of one or more intermediate points that can vary depending on the current workpiece's size and the current state of the work cell. An example program is shown in Listing 3 where P1 to Pn denote a sequence of intermediate points with Pn being the target point for the movement.

```
Move Robot to P1;
Close Gripper;
Move Robot to P2;
...
Move Robot to Pn;
Open Gripper;
```

Listing 3. Manipulator-level program, obstacle avoidance

Common solutions for this problem are either iterating over an array of points or using elaborate movement commands which accept a sequence of points as input (e.g. a spline).

C. Object-centric solution to the example

As shown previously, there exists a manipulator-level solution for each variation of the basic scenario. On the other hand, those solutions yield substantially different *programs* – though the actual *task* remains the same. It is a well-known fact in software engineering that encoding the (business) logic of the task and all additional requirements and variations in one program introduces a high coupling between all elements of the program, in this case between the control program and the workpieces that it can handle. As a consequence, the resulting software will be hard to maintain, reuse and extend.

This also holds for the robotic domain. A robot program that considers all those variations at once or even additional ones will obviously become very complex. With the object-centric approach, this problem is addressed. The key concept is programming on the level of the objects that are involved in the task that has to be achieved, and augmenting those objects with certain properties and functionalities. The goal is being able to handle variations in tasks efficiently by a modular design with reusable components.

```
Box1.Bottom.Move(Box2.Top);
```

Listing 4. Object centric program for stacking boxes

The pseudo code snippet in Listing 4 illustrates how an object-centric program looks like. First, there is a notion of workpieces, *Box1* and *Box2*, which represent the two boxes involved in the scenario. Workpieces are distinct entities that have certain properties and the ability to execute certain actions. Interesting properties of a workpiece would be a point where it can be gripped, and points located at its top, bottom and sides. An important ability of a workpiece is the execution of movements. In the example, this ability is addressed by the method *Move()*. This method belongs to *Bottom*, which itself is a property of *Box1* and represents a point located at the bottom of the box. Semantically the invocation of this method is the instruction to execute a movement, with the point *Box1.Bottom* as origin and *Box2.Top* (located on top of Box2) as the target of the movement.

Note that in Listing 4 *no* robots are addressed at all. The prominent concept in this example is the notion of domain- and task-specific objects that are not only passive data objects, but can actively react to instructions. This is a standard approach in modern software development. Standard graphic libraries e.g. offer classes like *Circle* or *Polyline*. These classes have specific implementations of methods like *Draw()* or *Move()*. The actual graphic board commands are hidden in the class-specific implementations.

The logic for executing such tasks and handling variations of tasks is inevitable, of course. But instead of encoding it as part of the task description as it is done in manipulator-level programming, the logic is distributed to objects that have all necessary information. For example, the *Move()* operation typically contains two sub-operations: choosing an adequate manipulator and subsequently giving appropriate commands to the manipulator. Both sub-operations are highly *workpiece-specific* and, therefore, are defined as a method of the according object. A workpiece or part of a workpiece *knows* how it is to be gripped and can find matching manipulators and generate matching commands for them.

Hence, the variations of the previously introduced example can be handled easily with the object-centric approach:

1) Dimensions or position: The size and the position of a workpiece are naturally modeled as properties of the workpiece. They can either be set by the programmer at design time or determined by sensors at runtime. These values are encapsulated within the workpiece (cf. *Box1.Bottom*) and can be considered when handling commands.

2) Shape and weight: Variations in the workpiece's shape can partially be reflected by its properties, for example regarding a different location of the point where the workpiece can be gripped by a manipulator. This data is processed within the *Move()* method. It will be an important input data for choosing a manipulator and also for generating movement commands for the manipulator. Possible solutions could apply mechanisms similar to service-oriented architectures. Here, a query to a service directory for a service fulfilling certain requirements is performed. Translated into the robotics domain, the workpiece has to query for one or more manipulators that – single or in sum – can solve the requirements for moving the workpiece. When adequate

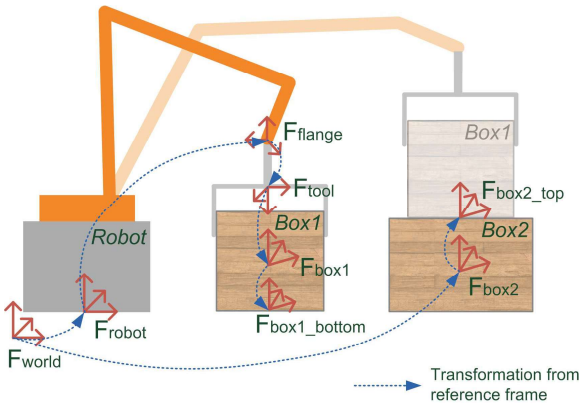

Fig. 2. Stacking boxes: entities and frames

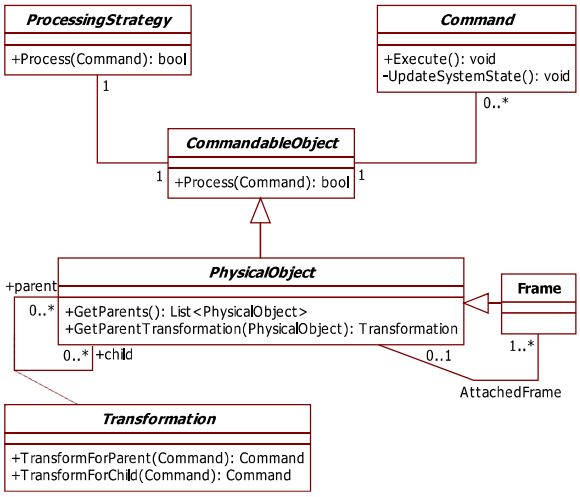

Fig. 3. Conceptual model enabling object-centric robot programming

manipulators have been found, the workpiece has to trigger the movement and gripping process for each manipulator. Finally, after the box is gripped, the synchronized movement must be started. Note, that the (workpiece-specific) logic is again encapsulated within the according object and the program describing the actual task stays untouched.

3) Environment and obstacles: Besides specifying *tasks* in a robotic system, the object-centric programming approach explicitly describes as a prerequisite *real world objects* like workpieces, pallets or any other objects of relevance. Together, this forms a partial model of the reality. Research has developed many algorithms for planning collision free movement paths. Such algorithms can be easily integrated into the logic that processes movement commands in the object-centric programming approach, and the existent world model can be used as input describing the environment.

III. Realization approach

The last section introduced the core concepts of an object-centric programming paradigm. The idea is putting tasks and target objects in the center of any robotic program, rather than the robots. This allows for much easier, more modular and more extensible programs. Building on that, this section presents a possible realization approach and a prototypical object-oriented framework for developing novel object-centric robotic applications.

A. Basic concepts for describing spatial properties

To every important spatial point of physical objects, a frame is attached, describing its position and orientation (cf. [10]). A frame F_x is a Cartesian coordinate system, defined as a tuple (F_{ref}, A_x^{ref}), where F_{ref} is the *reference frame* and A_x^{ref} is the *homogeneous transformation* used to gain F_x from F_{ref}. The inverse transformation is denoted with $(A_x^{ref})^{-1}$ or A_{ref}^x. In many cases, transformations are fix, but dynamic transformations that e.g. change over time are also possible by adequate parametrization. There is no restriction to the choice of the reference frame: A frame can be defined relative to any other previously defined frame. However, there exists one special frame F_{world} – the *world frame* – that is a globally unique anchor frame which all other frames must be based on directly or indirectly. By definition, the world frame is the only frame that does not have a reference frame. To determine the absolute position of a particular frame at a certain point of time, the complete transformation to F_{world} has to be calculated by recursively traversing the transformations to the frame's direct and indirect parents.

Any frame F_x is defined with respect to its reference frame F_{x-1} in the first instance. By utilizing the transformations between frames, F_x can be expressed relative to any other frame F_y. The precondition for that is that there exists a sequence of transformations A_{x-n}^y that *connects* F_x and F_y. With the definitions above, there exists for every frame a sequence of transformations connecting it with F_{world}. The dotted arrows in Fig. 2 indicate the relationships between the frames in the palletizing example. Each frame (except the world frame) has an incoming arrow starting at its reference frame.

B. Software architecture

In order to write object-centric programs as previously described, a specially designed control software is needed. This software must support the notion of arbitrary objects that are capable of receiving and processing commands depending on their current context. Additionally, it must be possible to express relationships among those objects (e.g. the geometric relationship shown in Fig. 2). It should be noted that real-time aspects, which certainly play an important role in low-level robot control, are not considered at this level. Instead, it is assumed that finally all basic commands can be delegated to a robot control system which executes the commands while

taking care of all real-time critical aspects. The architecture described in this section provides a framework for object-centric specification of robotic applications, and completely abstracts from the underlying robot control. Fig. 3 shows a conceptual model fulfilling this requirement.

The central concepts are *Commands* and *CommandableObjects*. A *CommandableObject* can be any entity that (directly or indirectly) performs actions relevant in a certain task. Examples are robots, tools or workpieces. Tasks are specified by *Commands*. As an abstract class it is the base for any concrete command and provides methods for execution. The approach of representing commands as distinct objects is a variant of the *Command* pattern [11] and has some advantages over implementing commands as methods like the pseudo code in Listing 4 indicated:

- Standard *Commands* can be defined once and used for many (new) *CommandableObjects*.
- *Commands* can encapsulate their status of execution.
- Each *Command* can be extended e.g. by undo/redo.
- Special macro commands can be introduced that aggregate a series of *Commands* that have to be executed as an atomic step in a serial or parallel manner.

Which *Commands* are accepted by a *CommandableObject* and how it interprets them depends on the type, but also on the current context of the object. For example, when a box should execute a movement command, it will only be able to execute that command if it can be gripped and moved by a manipulator. To enable this context-specific handling of commands, each *CommandableObject* relies on a *ProcessingStrategy* determining how entities interpret certain commands in their current context. The *ProcessingStrategy* of a *CommandableObject* is exchangeable during runtime, reflecting changes in the context of the object. This mechanism is an application of the *Strategy* pattern [11].

To represent spatial relationships, real-world physical objects are represented in the framework by *PhysicalObjects*. They can be part of a geometric hierarchy such that each *PhysicalObject* can have other *PhysicalObjects* as parent or child elements. An example for such a hierarchy is a robot that has a gripper holding a workpiece attached to its flange. The workpiece's parent element would be the gripper, which is a child of the robot's flange. To every parent-child-relationship, a *Transformation* is attached. By using this transformation, a *Command* intended for the *PhysicalObject* in the parent role can be transformed to a *Command* for the *PhysicalObject* in the child role (and vice versa). However, both *Commands* must still have the same interpretation for the system as a whole. In the example given above, the command telling the workpiece to move on a linear path can be transformed into a movement command for the gripper holding the box (and finally into a command for the robot). Hence, the transformed command must describe a movement of the gripper (or rather the robot) that results in the movement of the workpiece as it was specified by the original command.

The concept of a *Frame* is explicitly included in the concept model, as it is intended to be a central instrument for describing important parts of physical objects. Each *PhysicalObject* has at least one *Frame* attached to it identifying its location in space. Note that a *Frame*, being a *PhysicalObject*, is also a *CommandableObject* and thus can receive *Commands*. This enables writing programs like in Listing 4 where the command is actually given to a *Frame*.

C. Implementation and dynamic aspects

Fig. 3 depicts the static structure of the software design. To illustrate dynamic sequences of actions, the implemented prototype is explained below. The implementation uses the object-oriented language C# that is part of Microsoft's .NET framework [12]. For testing and visualization, Microsoft Robotics Developer Studio 2008 [13] was utilized. It is a service-oriented programming environment for robotic devices that incorporates a visual simulation environment with an integrated physics engine (see Fig. 1). The simulation environment greatly served the purpose of building a lightweight prototype by supporting the creation of a visual and physical model of the scenery.

The simple box-stacking example introduced in Sect. II was implemented using the object-centric architecture. The robot, its gripper and the boxes are modeled as *PhysicalObjects* with the gripper being a child of the robot. We assume that the box is already gripped. During the gripping, the gripper became the parent of the box, and a special processing strategy called *DelegationStrategy* was associated with the box. This strategy assumes that the *CommandableObject* cannot process *Commands* itself. Therefore, it tries to find a manipulator capable of executing the movement command by delegating the command to an object's direct parent. It uses the *Transformation* to transform the *Command* such that the semantics is preserved when it is delegated. The parent will then decide, based on his own processing strategy, how to further process the *Command*.

Figure 4 shows the delegation process in detail. The issuer of the linear movement command *cmd* of type *LinCommand* invokes the method *Execute()* to trigger execution. Then, the command delegates the processing to the object the command was intended for – the *Frame* called *bottom* (cf. step 2). However, the actual processing of the command is done by a *DelegationStrategy*. It retrieves the parent elements of the frame and picks the first one. It is assumed here that each *PhysicalObject* has only one parent. The next step is obtaining the *Transformation* for the parent (cf. steps 7 and 8). The transformation is a *GeometricTransformation*, which uses the information about the spatial relationship between parent and child to transform movement commands. Finally, the *DelegationStrategy* calls the *Process()* method of

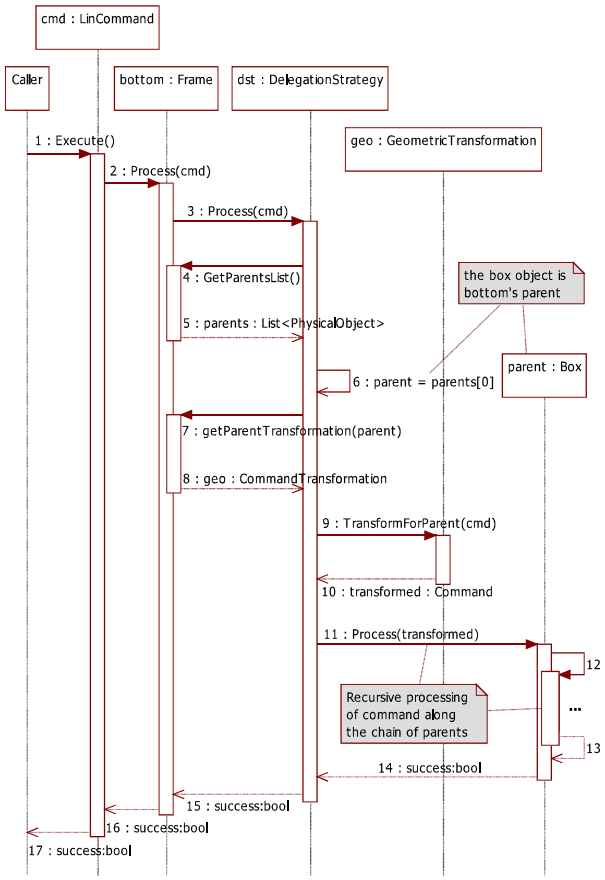

Fig. 4. Delegation of a Command

its *PhysicalObject*'s parent (cf. step 11) which does the same internal processing of the (now transformed) *Command*. This recursive internal processing determines how the command is interpreted in the end (or if it is even handled at all). This mechanism can be seen as an extension of the pattern *Chain of Responsibility* [11]. By combining this pattern with the afore mentioned Strategy pattern, the way how each node in the chain (i.e. the parent-child-relationship) treats commands can change dynamically.

IV. Conclusion & Future Work

This paper introduces a new programming paradigm for industrial robots called *object-centric programming*. Object-centric robot programs are basically the consequent application of object-orientation and modern software engineering paradigms to the domain of industrial robotics. This enables a systematic translation from domain problems into robot programs, which are automatically transformed into manipulator instructions. As a consequence, software development for industrial robots will become easier and more efficient.

While there exists a number of object-oriented robotic frameworks, they mainly focus on providing real-time control structures and do not abstract from manipulator-level programming. They could, however, be used for implementing a real-time capable robotic control core that processes basic commands to robots, tools and other devices. Examples are MRROC++ [4], ZERO++ [14], SIMOO-RT [15] and OROCOS [16].

The software architecture introduced in Sect. III enables the programming of object-centric applications. With this design, various objects can handle commands depending on their current context, which allows a reduction of coupling between robot control programs and specific objects they operate on. Moreover, the approach was tested by implementing a prototype using the object-oriented language C# and Microsoft Robotics Developer Studio. While the implemented scenario only covers parts of possible task variations that were discussed in this work, the architectural design allows for a straightforward handling of other variants. Besides extending the prototype in this direction, current work also includes the integration of a KUKA LBR robot into the framework. First results are very promising. Further research is done to develop mechanisms for specifying coordination between dependent commands, like in the case of load sharing with multiple robots.

References

[1] *Programming Handbook (Release 5.2)*, KUKA Robot Group, 2004.
[2] *RAPID Reference Manual*, ABB Group, 2004.
[3] FANUC Robotics. FANUC Robotics America Inc. [Online]. Available: http://www.fanucrobotics.com/
[4] C. Zieliński, "Object-oriented robot programming," *Robotica*, vol. 15, no. 1, pp. 41–48, 1997.
[5] R. J. Popplestone, A. P. Ambler, and I. Bellos, "RAPT: A language for describing assemblies," *Industrial Robot: An International Journal*, vol. 5, pp. 131 – 137, 1978.
[6] C. Blume and W. Jakob, *Programming Languages for Industrial Robots*. Springer-Verlag, 1986.
[7] C. Zieliński, "TORBOL: An object level robot programming language," *Mechatronics*, vol. 1, no. 4, pp. 469–485, 1991.
[8] A. Cockburn, *Writing effective use cases*. Addison-Wesley, 2001.
[9] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed. Addison-Wesley, 1994.
[10] J. Craig, *Introduction to Robotics: Mechanics and Control*, 3rd ed. Prentice Hall, 2005.
[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[12] .NET Framework Developer Center. Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/netframework/
[13] Microsoft Robotics Studio Developer Center. Microsoft Corporation. [Online]. Available: http://msdn.microsoft.com/robotics/
[14] C. Pelich and F. M. Wahl, "ZERO++: An oop environment for multiprocessor robot control," *International Journal of Robotics and Automation*, vol. 12d, no. 2, pp. 49–57, 1997.
[15] L. B. Becker and C. E. Pereira, "SIMOO-RT – An object oriented framework for the development of real-time industrial automation systems," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 421–430, August 2002.
[16] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, Seoul, Korea, May 2001, pp. 2523–2528.