Object-Oriented Techniques in Robot Manipulator Control Software Development

Markus S. Loffler, Darren M. Dawson, Erkan Zergeroglu and Nicolae P. Costescu

Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634-0915
[loffler, ddawson, ezerger, ncostes]@ces.clemson.edu

Abstract

Software development for the control of robotic manipulators is a complex task because it requires expertise in many areas (e.g., robotics, real-time programming, hardware integration, concurrency, etc). Because of this fact, it is difficult to develop a common software platform supporting the diversity of robotic research areas and robotic hardware. Even though a large number of robotic languages, libraries, and tools have been created, they are seldom reused. That is, many research teams develop their own software platform from scratch because existing platforms are too inflexible with regard to modifications and too complex to understand. The authors of this paper believe that code reuse is fostered by providing a lightweight platform rather than requiring the user to modify a complex system. Along this line of reasoning, this paper describes the QMotor Robotic Toolkit (QMotor RTK). The RTK is a set of C++ libraries and programs that follow object-oriented concepts to ensure code reuse, modularity, scalability, and an intuitive code structure. The RTK is a homogeneous system that consists only of PC software. The RTK includes joint level control programs for the Puma 560 manipulator, the Barrett Whole Arm Manipulator (WAM), and the Integrated Motion Inc. (IMI) two-link manipulator as well as a joint level trajectory generator and a graphical user interface (GUI).

1 Introduction

Software development for the control of robotic manipulators is a complex task. That is, even for simple pick and place operations, the software developer needs to incorporate hardware interfacing, concurrency, real-time programming, servo control programming, and trajectory generation into the software platform. In more advanced applications, manipulators operate based on sensor and visual feedback (*e.g.*, to implement visual servoing and force based control). Finally, many modern systems provide advanced operator consoles based on visual feedback, virtual reality, and a graphical user interface. To minimize the development effort, it is desired to build on an existing software platform. However, the diversity of robotic research areas, applications, and robotic hardware has not fostered the development of a commonly used platform.

The necessity for such a platform can be understood by looking at our previous work regarding the utilization of robotic manipulators for decommissioning tasks [1][2][3]. Our disassembly system first utilized the robot control library RCCL [4] and a Puma 560 robot. This structure was not flexible enough to implement more complex controller types, as required in disassembly operations, because the servo level was implemented on a proprietary Mark II controller. To increase flexibility, we developed a servo control program executing on the QNX 4 real-time operating system [5] along with an interface to the robot control library RCCL. Subsequently, due to the limitations of this system, we started over again and ported ARCL [6] to QNX 4. Although at all stages of the project, the result was a working platform [2][3], each platform had several disadvantages. First, a lot of development time had to be spent in areas that were not really part of the research issue: Porting libraries, writing interfaces between different software packages, studying of code written by others, *etc.* Second, the system was very proprietary to the application and the hardware environment. For example, later in this project we wanted to integrate the WAM [7] into our disassembly system. This modification would require a lot of effort because it would involve the development of a complete new servo control program for the WAM as well as modifications to ARCL and all GUI programs. Finally, the system did not provide any means for control tuning and data logging of the servo control program.

The experience of this disassembly resulted in the formulation of the following requirements for a reusable software platform:

- *Flexibility*. The platform should be easily extensible for new components, especially for new manipulators. Modifications and extensions of the platform should be possible on all levels of the system (*i.e.*, task level, trajectory generation level, and servo control level).
- *Real-Time Support*. The platform should provide support for real-time operations. In addition, the user should be able to debug the real-time code, log and plot control signals, and tune the controller.
- *Modularity*. The platform should be structured into components that can be easily added and reconfigured. Also, researchers are often interested in just one special component of the platform (*e.g.*, they are interested in improving the servo control algorithm). Hence, modularity allows the researcher to focus on his interest without learning the internals of the rest of the platform.

2 Previous Systems and Research

To implement a robotic system, developers often utilize: i) robot control languages, ii) common programming languages like C/C++, iii) graphical control environments, and iv) robotic libraries for common programming languages.

Robot control languages provide a set of commands for implementing the control application. They are usually provided by the manipulator vendor and custom tailored to the specific manipulator type. Additionally, they are often based on proprietary hardware (*i.e.*, special purpose processors). Many of these languages do not allow modification (*e.g.*, implementing new control strategies) and extension (*e.g.*, interfacing to new system components such as sensors, visual feedback, *etc*). Hence, the scope of proprietary robot control languages is limited.

The most direct method for developing a software platform is to implement a new solution from scratch, in a common programming language such as C. The advantage of this approach is that one is able to design the system in a way that fits exactly his needs and requirements. However, there are many disadvantages to this approach. Specifically, due to the complexity of the problem, development is very time consuming, error-prone, and requires a high level of skill.

To cut down on development time, solutions are available that simplify control development on the servo level. For example, QMotor [8][9] is a graphical control environment that requires the use of C/C++ to implement the control algorithm, but takes care of the programming issues related to timing, control tuning, data

logging, and plotting. There are also several software platforms available that are based on MATLAB/Simulink, allowing the developer to create block diagrams instead of implementing the control as a C/C++ program. Real-Time Linux Target (RTLT) [10], Real-Time Windows Target [11], and WinCon [12] are examples of this concept. However, even though it is possible to implement a manipulator control system as a block diagram, the required functionality very often leads to complex block diagrams. In addition, the use of Simulink limits hardware related functionality and greatly increases the computational burden on the real-time platform (*i.e.*, as opposed to developing a C/C++ program).

Unfortunately, the development of a manipulator software platform, even when supported by the above-described environments, is an extensive task. That is why software libraries have been developed that provide data types and functions for robotic applications. The most well known example is RCCL [4]. The robot control library ARCL [6] is less complex and less powerful than RCCL, but follows the same concept. However, there is no straightforward way to modify the servo control level in RCCL and ARCL (*e.g.*, for Puma 560 robots, the servo control runs on a proprietary Mark II controller); therefore; it is not straightforward to implement new control strategies. Also, the large amount of code and complexity of RCCL and ARCL make them very difficult to understand and modify. RCCL and ARCL are good examples of procedural programming reaching its limits. That is, both libraries use programming constructs (*e.g.*, function pointers) that emulate object-oriented concepts. However, since the implementation language (C) is not object-oriented, these constructs are difficult to understand and modify.

All of the solutions described above have one common problem. Specifically, if new functionality is needed or if new hardware is required, one must modify the source code (or the block diagrams, respectively). Modification of the systems internals is very error-prone. To overcome this problem, there have been object-oriented approaches to robot control libraries. For example, RIPE [13], developed at Sandia National Laboratories, defines an intuitive hierarchy of classes for robotic hardware. However, RIPE does not use object-oriented concepts at the servo level. MMROC+ [14] uses an object-oriented design for error handling and simplification of the communication between processes. OSCAR [15] is an extensive library that addresses many issues of object-oriented design for robotic systems. It focuses mainly on the operational software layer (the layer between the user interface and the servo control). However, it is also very complex and requires multiple computing platforms.

3 The QMotor RTK System

The QMotor RTK system is structured as a combination of ready-to-execute programs and C++ libraries. The system is implemented on QNX 4 and QNX 6 (the QNX Real-Time Platform), which are very reliable real-time operating systems. To avoid addressing timing, data logging, and plotting, the real-time control environment QMotor [8][9] is used as the base for the RTK. QMotor allows object-oriented control implementation in which control programs can be implemented as C++ classes. The RTK takes advantage of this concept and builds on the QMotor classes.

The QMotor RTK works only at the joint level, (*i.e.*, forward/inverse kinematics and Cartesian trajectory generation are not included). The RTK contains joint level position control programs for the WAM, the Puma 560, and the IMI manipulator. Also included is a generic joint level trajectory generator and GUI based teachpendant. Additionally, various utility programs are part of the RTK. The object-oriented approach is used in the control development as well as for the GUI components.

Figure 1 shows a typical QMotor RTK configuration. Each box represents a separate program. Lines represent message paths between the programs. The example system contains the teachpendant, the trajectory generator, the WAM servo control, and the WAM control panel. A ServoToGo S8 motion control board provides the hardware interface to the manipulator. To reconfigure the system, one only has to start different programs. For example, for the replacement of the WAM by a Puma 560 robot, one would start the program "`pumacontrol`" instead of "`wamcontrol`".

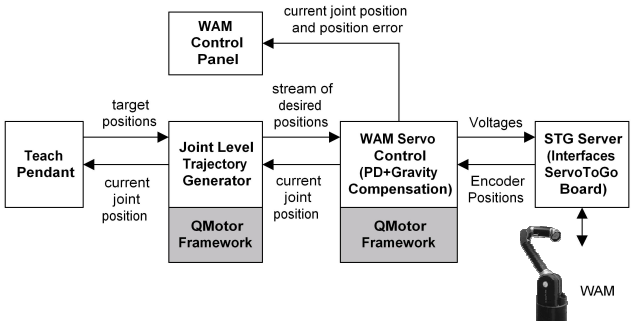

Figure 1. A Typical QMotor RTK Configuration

4 Data Logging, Plotting, and Control Tuning in an Object-Oriented Environment

As the system design discussed in this paper starts by implementing the servo control loop, one needs to implement a cycling control loop. Furthermore, it is desired to offer functionality with regard to tuning the control algorithm and gathering data during the control run. These tasks are not trivial to implement. They also add large overhead to the programming effort. Thus, it is helpful to reuse existing software to save development time. The QMotor environment is well-suited for all real-time components of the RTK. QMotor is a general real-time environment for the development of any kind of control program. It contains three components: Hardware servers, the control program library and the QMotor GUI.

The *Hardware Servers* provide a generic interface to motion control boards and other hardware components. Currently, the QMotor RTK utilizes hardware servers for the MultiQ and the ServoToGo S8 motion control boards.

The *Control Program Library* is utilized to implement the control algorithm as a C/C++ program. A base class, called `ControlProgram`, contains the framework for implementing control programs. The user derives a class that is specific to the control application from the `ControlProgram` class (*e.g.*, `ManipulatorControl`) and fills in the necessary functionality to implement the control algorithm. This functionality is contained in five functions that are left blank in the base class `ControlProgram`:

- `enterControl()`: Called when the control program is loaded
- `startControl()`: Called every time the control execution is started
- `control()`: Called regularly at the control frequency
- `stopControl()`: Called when the control execution is stopped

- `exitControl()`: Called when the control program terminates
- `handleMessage()`: This function allows the control program to perform as a server, since `handleMessage()` is called when a message from another task (*i.e.*, the client task) arrives.

The *QMotor GUI* is used for selecting logging options, for plotting signals, and for control tuning. C++ variables of the control program can be registered as control parameters to give the user the ability to change them from the GUI environment. In the same fashion, other C++ variables can be registered as log variables, thereby, making them available to be logged and plotted in the GUI. Figure 2 shows a QMotor plot window.

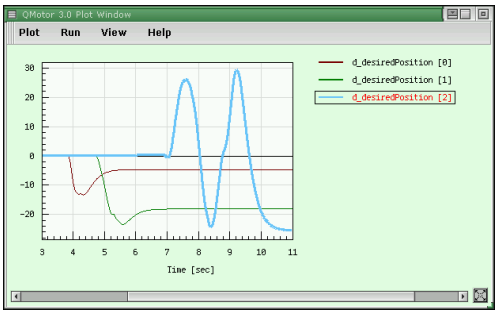

Figure 2. A QMotor Plot Window

5 Object-Oriented Design

The procedural programming approach is based on two major concepts:

1. Data representation (*e.g.*, representation of the current position error of a manipulator).
2. Functions that operate on this data (*e.g.*, a function that calculates the required torques from the position error).

The above two concepts exist in the object-oriented approach as well. However, while procedural programming treats them separately, the object-oriented design ties them together. That is, they are grouped together in a construct called a *class*. There can be any number of classes in the system, identified by class names. For example, a `PumaControl` class would contain all of the data related to the control of a Puma robot (*e.g.*, current position, desired position, output torques, *etc.*) and all functions that are related to the control (*e.g.*, calculate the control algorithm, enable the arm power, *etc.*). To design an object-oriented system, the software engineer must carefully group data and functions in classes. Considering a software platform for robotic applications, this choice is intuitive. For example, classes represent physical objects such as the manipulator. Additionally, there are classes that represent functional objects (*e.g.*, the trajectory generator) and classes for GUI components. Consequently, the use of classes leads to a very intuitive modeling of the system.

There are several useful programming techniques utilized in object-oriented programming: i) data abstraction, ii) encapsulation, iii) polymorphism, and iv) inheritance [16]. Among other benefits, these programming styles have the following advantages:

- To use a class, an *object* of the class has to be instantiated. To operate multiple physical objects (*e.g.*, to control two manipulators of the same kind), the programmer simply instantiates multiple objects of the same class.
- *Polymorphism* is the ability to provide the same interface to objects with differing implementations. Polymorphism is useful for developing generic programs (*e.g.*, a trajectory generator can use the same generic interface for different manipulators).
- The use of classes leads to an open system that allows extension of the system via the design of new classes. Specifically, *inheritance* can be utilized. That is, any class can be defined to reuse generic data and functions from another class.

The idea of *inheritance* with regard to manipulator control software is now examined in detail. Once one starts to design classes for a manipulator control system, similarities between these classes become apparent. A class for a Puma 560 robot and a class for the WAM contain common functionality (*e.g.*, they both utilize a servo control algorithm, receive a desired trajectory, determine the current position by encoders, *etc.*).

A simple approach to develop both classes would be to first develop the class for the Puma 560 robot and then either rewrite the code for the WAM or copy the Puma 560 code and modify it (see Figure 3a). However, this approach leads to additional development effort; and hence; a higher probability of new errors. In addition, if the common functionality changes (*e.g.*, due to bug fixes or improvements), then changes need to be applied to all of the copies.

To avoid these disadvantages, the inheritance feature of object-oriented programming can be utilized. To use inheritance, a base class `ManipulatorControl` is defined. This base class contains the common functionality as described above. Then, the specific classes for the Puma 560 and the WAM manipulator are *derived* from this base class (see Figure 3b). Deriving means that they take over the functionality and data from the base class. Additionally, they are also able to redefine parts of this functionality and/or add new functionality and data. Once the base classes have been developed, they do not need to be re-compiled when a new derived class is added. That is, one does not need to change any source code of the base class to extend the system. On the other hand, a modification of the common functionality in the base class is automatically reflected in all derived classes. Hence, inheritance greatly supports code reuse.

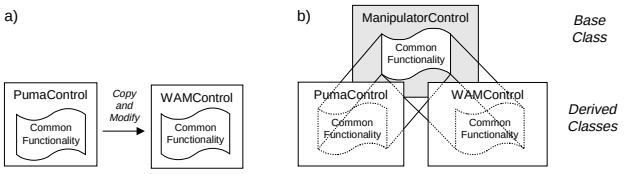

Figure 3. Code Reuse through a) Code Duplication, and b) Object-Oriented Programming

6 Design of the Manipulator Control Classes

6.1 The Base Class

The lowest level of the QMotor RTK is the servo control level. This level consists of an independent PD joint tracking controller and the interface between the computer and the robot via a motion control board. The servo control level is implemented for three different manipulators: The Puma 560 robot, the WAM, and the IMI robot. As mentioned earlier, the first step in object-oriented design is to distinguish between common functionality/data and

specific functionality/data. This concept is illustrated for the servo control level in Table 1 and Table 2.

Table 1. Common and Specific Functionality for the Manipulator Control

Common Data	Specific Puma Data
• Joint position, velocity, acceleration for n joints • Control gains • Control modes • Joint and torque limits • Variables for I/O board control • Other control parameters	• Potentiometer values
	Specific WAM Data
	• Torque ripple data
	Specific IMI Data

Table 2. Common and Specific Data for the Manipulator Control

Common Functionality	Specific Puma Functionality
• Communication with the motion control board • Setting output torques by setting voltages of the digital to analog converters (DACs) • Position readings through encoders • Enabling/disabling arm power by setting digital outputs • PD position control • Determining velocities and accelerations by backwards difference and filtering • Communication with client tasks (*e.g.*, to receive a desired trajectory) • Switching between control modes (*e.g.*, zero gravity mode/position control mode) • Safety checks for joint and torque limits • Manual calibration • Generation of a simple test mode trajectory	• Automatic encoder calibration • Motor angles to joint angles transformation (to include coupling effects) • Gravity compensation
	Specific WAM Functionality
	• Automatic encoder calibration • Motor angles to joint angles transformation (to include coupling effects) • Joint torques to motor torques transformation • Gravity compensation • Torque ripple compensation
	Specific IMI Functionality
	• Disable arm power functions (There is no software control over the arm power)

All common functionality (Table 1, left column) and data (Table 2, left column) is contained in the base class `ManipulatorControl`. This class, which is derived from the `ControlProgram` class, implements all the QMotor functions of the `ControlProgram` class that were left empty (*i.e.*, `enterControl()`, `exitControl()`, `startControl()`, `stopControl()`, `control()`, and `handleMessage()`). All functions that should be available for reimplementation in a derived class are declared as *virtual functions*. That is, even if such a function is called from the base class, the reimplemented function will be used.

Some functions of the base class `ManipulatorControl` contain common functionality, while others are just left empty (*e.g.*, the `doCalibration()` function is responsible for the automatic calibration procedure, and hence, is highly manipulator dependent). In the derived classes for the Puma 560 robot, the WAM and the IMI robot, new functions are added and certain functions are reimplemented with modified functionality, as listed in Table 1 and Table 2. Since the major part of the work is done in the base class `ManipulatorControl`, the derived classes are significantly smaller and simpler.

6.2 The PumaControl Class

The following extensions are made in the `PumaControl` class:

- Variables and functions for the automatic encoder calibration procedure are added. This procedure determines the absolute position of the Puma by first getting a rough estimate from potentiometer readings and then performing the calibration by searching for the next index pulse.
- The function `getCurrentPosition()` is reimplemented to take the coupling of joints 4, 5 and 6 into account.
- Gravity compensation is added. Gravity compensation calculates the torques resulting from the manipulator's weight and adds these to the output torque for compensation [17].

6.3 The WAMControl Class

The following extensions are made in the `WAMControl` class:

- Variables and functions for the automatic encoder calibration procedure are added.
- The functions `getCurrentPosition()` and `setControlTorque()` are reimplemented to take the coupling of joints 2/3 and joints 5/6 into account.
- Gravity compensation is added.
- Torque ripple compensation is added.

The automatic calibration procedure of the RTK determines the absolute position of the WAM by moving joint by joint to its joint limits. The joint limit is detected by the position error exceeding a certain threshold. Then, a weighted sum of the encoder values at the minimum and the maximum joint limit determines the zero position of the WAM. This procedure is a somewhat lengthy operation; however; it is necessary because the WAM does not contain hardware to determine its absolute position (*e.g.*, potentiometers).

For the gravity compensation, the WAM is modeled as three point masses. Two of these point masses are located at the center of mass of each link, and the third is located at the end of the robot arm. Lagrange's equation of the manipulator is simplified by the static conditions of the manipulator holding the position (*i.e.*, the joint velocities and the kinetic energy are zero). This simplified equation can be used to calculate the required torques [7]. To determine the mass parameters of the equation, a calibration procedure is implemented as a separate program. This program, called "gravity calibration utility", moves the WAM with the position control (without gravity compensation) to three predefined positions and measures the average torque to hold the WAM at this position. From those torque values, the mass parameters can be calculated.

Finally, a small signal is added to the motor control voltage to compensate for the torque ripple of the electric motors to provide smoother movement in the zero gravity mode.

6.4 The IMIControl Class

The only modification in the `IMIControl` class concerns the arm power functions. As the IMI does not have arm power control by software, the arm power functionality is removed in the derived class `IMIControl`.

7 The GUI Components

The design of GUI components is very important with regard to simplifying the use of the manipulator control system. Real-time operating system like QNX 4 and QNX 6 allow GUI programs to coexist with high priority control programs. The RTK contains four GUI programs: the manipulator control panel, the WAM control panel, the manual-move utility, and the teachpendant.

QWidgets++ [8] is an object-oriented library for GUI programming under QNX. QWidgets++ was selected for the GUI programs of the RTK because it facilitates a pure object-oriented design. Specifically, GUI elements (*e.g.*, buttons, windows, *etc.*), also called widgets, are represented by C++ classes. The manipulator control panels demonstrate the use of inheritance at the GUI level. The manipulator control panel (see Figure 4) is a generic control panel that works with all manipulators. The WAM control panel has two additional buttons to control torque ripple compensation and manual calibration (Figure 5). To avoid duplicating the common features of both control panels, a base class `ManipulatorControlPanel` is created that implements the common features of both control panels. The class `WAMControlPanel` is then derived from the class `ManipulatorControlPanel` to add the additional buttons to the control panel window.

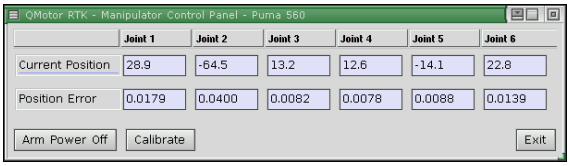

Figure 4. The Generic Manipulator Control Panel

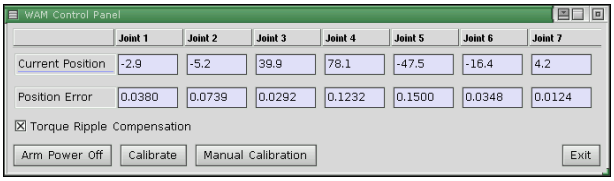

Figure 5. The WAM Control Panel

The teachpendant (see Figure 6) uses the zero gravity mode of the manipulator to allow the user to push the manipulator around in the workspace. Once the user has moved the manipulator to a desired target position, this position can be added to a list of points. The teachpendant also utilizes the trajectory generator to move the manipulator back to the taught positions. Additionally, the teachpendant is able to control the Barrett Hand, an advanced three finger gripper. Hence, complete pick and place operations can be programmed with the teachpendant.

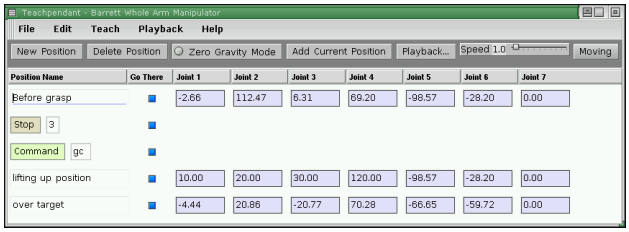

Figure 6. The Teachpendant

8 Modifying the System Using Inheritance

The previous sections explained how object-oriented techniques accelerate the addition of new components to the QMotor RTK. This section illustrates in greater detail how inheritance can be used during the addition of a new control algorithm. Specifically, in this simple example, the controller is extended from the PD controller to a PID controller.

Figure 7 shows the function `calculatePositionControl()`, which calculates the PD control. This function is part of the class `ManipulatorControl`. To implement the new controller, a new class `WAMPIDControl` is derived from the class `WAMControl` (see Figure 8, [a]). This class reimplements the function `calculatePositionControl()`. The reimplemented function first calls the `calculatePositionControl()` function of the base class; and hence, uses the algorithm for the PD control of the base class (see Figure 8, [b]). Then, the integral term is added (see Figure 8, [c]). Note that the function `calculatePositionControl()` of the base class and the derived class are distinguished by the scope prefixes "`ManipulatorControl::`" and "`WAMPIDControl::`".

```
void ManipulatorControl::calculatePositionControl()
{
  // PD control plus acceleration feedforward
  for (int i = 0; i < d_numJoints; i++)
  {
    d_controlTorque[i] += d_kp[i] * d_positionErrorRad[i]
     + d_kd[i] * (d_desiredVelocityRad[i] - d_velocityRad[i])
     + d_feedforwardAccelerationGain[i] *
        d_desiredAccelerationRad[i];
  }
}
```

Figure 7. The PD Control Calculation in the Base Class

```
// ===== Class declaration of the derived class WAMPIDControl

class WAMPIDControl : public WAMControl                    [a]
{
  // ----- Constructors -----
public:
  WAMPIDControl (int argc, char *argv[])
     : WAMControl(argc, argv) {}
  ~WAMPIDControl () {};

  // ----- Manipulators -----
  virtual void calculatePositionControl();

  double d_ki[7];                    // Integral Gain
  double d_prevPositionErrorRad[7]; // Position error of the
                                     // previous control cycle
  double d_positionErrorInt[7]; // Integrated position error
};


// ===== Reimplemented function calculatePositionControl()

void WAMPIDControl::calculatePositionControl()
{
  // Call the base class to do the PD control
  ManipulatorControl::calculatePositionControl();          [b]

  // Then add the integral term
  for (int i = 0; i < d_numJoints; i++)                    [c]
  {
     // Integrate the position error
     d_positionErrorInt[i] += 0.5 * d_controlPeriod
       * (d_positionErrorRad[i] + d_prevPositionErrorRad[i]);
     d_prevPositionErrorRad[i] = d_positionErrorRad[i];

     // Add the integral term to the control torque
     d_controlTorque[i] += d_ki[i] * d_positionErrorInt[i];
  }
}
```

Figure 8. The Derived Class WAMPIDControl

9 Conclusions

This paper presented an object-oriented design for a software platform for robotic applications. Because of the complexity of large-scale software environments, the QMotor RTK was designed to be a lightweight modular platform. The RTK is a homogeneous, object-oriented system that is purely implemented as PC software. It utilizes a bottom-up design that is open and extensible from the servo-level to the task level.

The QMotor RTK reuses code for implementing different manipulator control programs and GUI programs. Specifically, base classes and classes for the Puma 560 robot, the WAM, and the IMI robot have been developed. Figure 9 relates the code size of the common and specific RTK components to the total code size. It illustrates that the implementation of new manipulators require a significantly smaller coding effort once the common base class is implemented. Note that a smaller coding effort also means a smaller source of coding errors. All new manipulator classes can refer to the well-tested base classes. The WAM control class, for example, had been developed without the manipulator present. Large parts of the RTK had already been tested with the Puma 560 robot. After the WAM arrived in the laboratories of Clemson University, the control program for the WAM was debugged and tuned within three days. The IMI control program was implemented, debugged, and tuned in a single day.

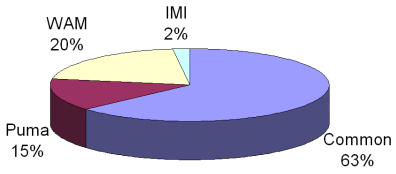

Figure 9. Code Size Ratios for the Supported Manipulators

We have also illustrated how object-oriented principles can be utilized to extend the system for new control algorithms shown with the example of a PID control. This example demonstrates one of the primary advantages of the RTK design: Coding effort of extensions is significantly smaller compared to implementation from scratch. Additionally, there is no need to modify source code when extensions are needed.

Future research is directed towards a new design that allows further simplification and a larger functionality while still maintaining a lightweight design. A more complete class hierarchy including sensors and tools is desired as well as classes for Cartesian control. Finally, a three-dimensional graphical simulation of the system's components would improve the system's testing capabilities significantly by adding off-line programming and testing.

References

[1] DOE Grant DE-FG07-96ER14728, "Advanced Sensing and Control Techniques to Facilitate Semi-Autonomous Decommissioning of Hazardous Sites", http://ece.clemson.edu/iaal/doeweb/doeweb.htm

[2] N. Costescu, M. Loffler, E. Zergeroglu, D. M. Dawson, "QRobot - A Multitasking PC Based Robot Control System", Microcomputer Applications Journal Special Issue on Robotics, Vol 18 No. 1, pages 13-22.

[3] M. Loffler, N. Costescu, E. Zergeroglu, and D. Dawson, "Telerobotic Decontamination and Decommissioning with QRobot, a PC-Based Robot Control System", Proc. of the Conference on Control Applications, Anchorage, AK, Sept., 2000, pp. 24-29.

[4] J. Lloyd, M. Parker and R. McClain, "Extending the RCCL Programming Environment to Multiple Robots and Processors", Proc. IEEE Int. Conf. Robotics & Automation, 1988, pp. 465 – 469.

[5] QSSL, Corporate Headquarters, 175 Terence Matthews Crescent, Kanata, Ontario K2M 1W8 Canada, Tel: +1 800-676-0566 or +1 613-591-0931, Fax: +1 613-591-3579, Email: info@qnx.com, http://qnx.com.

[6] P. Corke and R. Kirkham, "The ARCL Robot Programming System", Proc. Int. Conf. Robots for Competitive Industries, Brisbane, Australia, pp. 484-493.

[7] BA4-310 Software User Manual, Barrett Technologies, 139 Main St, Kendall Square, Cambridge, MA 02142, http://www.barretttechnology.com/robot.

[8] Quality Real-Time Systems, LLC., 6312 Seven Corners Center, Falls Church, VA 22044, Website: http://qrts.com.

[9] N. Costescu, M. Loffler, M. Feemster, and D. Dawson, "QMotor 3.0 – An Object Oriented System for PC Control Program Implementation and Tuning", Proc. of the American Control Conference, Arlington, VA, June, 2001, to appear.

[10] Zhigao Yao, Nicolae P. Costescu, Siddharth P. Nagarkatti, and Darren M. Dawson, "Real-Time Linux Target: A MATLAB-Based Graphical Control Environment", Proc. of the IEEE International Symposium on Computer-Aided Control Systems Design, Anchorage, AK, Sept., 2000, pp. 173-178.

[11] The MathWorks, 3 Apple Hill Drive, Natick, MA 01760-2098, http://www.mathworks.com.

[12] Quanser Consulting, 102 George Street, Hamilton, Ontario, CANADA L8P 1E2, Tel: 1 905 527 5208, Fax: 1 905 570 1906, http://www.quanser.com.

[13] D. J. Miller and R. C. Lennox, "An Object-Oriented Environment for Robot System Architectures", IEEE Control Systems February 1991, pp. 14-23.

[14] C. Zieliński, "Object-oriented robot programming", 1997, Robotica volume 15, Cambridge University Press, pp. 41-48.

[15] Chetan Kapoor, "A Reusable Operational Software Architecture for Advanced Robotics ", Ph.D. thesis, University of Texas at Austin, December 1996.

[16] B. Stroustrup, "What is 'Object-Oriented Programming'? ", Proc. 1st European Software Festival. February, 1991.

[17] B. Armstrong, O. Khatib, J. Burdick , "The Explicit Dynamic Model and Inertial Parameters of the PUMA 560 Arm", Proc. IEEE int. conf. Robotics and Automation 1, 1986, pp. 510-518.