A Reusable Software Framework for Rover Motion Control

Mihail Pivtoraiko, Issa A.D. Nesnas, Hari D. Nayar
Jet Propulsion Laboratory, California Institute of Technology
{mpivtora, nesnas, nayar}@jpl.nasa.gov

Abstract

This article describes the motion control interface for the Coupled Layer Architecture for Robotic Autonomy (CLARAty). This interface is used for controlling motors and other actuators in rovers and manipulators. Its development was initiated to address the needs of NASA-supported research projects, while the resulting design is general and can be applied to other systems. This paper lists and motivates the principal requirements for the interface, accumulated over more than seven years of the project development to date. A design of the motor interface that satisfies these requirements is described in detail. The key benefits of the design include ease of learning, using and maintaining, as well as significant code reuse.

1. Introduction

CLARAty is an object-oriented software infrastructure for the development and integration of new robotics algorithms, primarily for the use on the rovers [1]. Its purpose is to provide a common interface to a number of heterogeneous robotics platforms. This provides several important benefits, including simplifying the initial development and reuse of robotics algorithms in such areas as machine vision, manipulation, navigation, and planning [2]. CLARAty currently supports research rover platforms developed and used at the Jet Propulsion Laboratory, Ames Research Center, Carnegie Mellon University, and the University of Minnesota. In this paper, we focus on a part of this infrastructure that is responsible for designing and executing motions of actuators. Below we define the software engineering problem at hand and situate this work with regard to the state of the art. Further, we describe the requirements of software interfaces for motion control, followed by the details of a specific design that implements those requirements.

1.1. Problem Statement

Here we propose a solution to the problem of designing a software interface that requires satisfaction of three important properties:

- supporting and communicating directly with the physical embodiments of the controlled actuators (referred to as *hardware motors*),
- featuring high-efficiency to allow real-time execution on physical systems (Figure 1),
- maximizing software reuse across hardware systems and across software applications.

Designing good software interfaces that completely satisfy any one of these requirements is challenging. The CLARAty framework requires satisfaction of all three, thus making the present endeavor a hard problem. Here we propose a software Application Programming Interface (API) to motion hardware that leverages our experience in this framework and addresses the desired design properties.

Figure 1. Research prototype rovers contain many motors that must be controlled in a coordinated fashion.

1.2. Prior Works

CLARAty has included a motion control interface since its inception in 1999. Initial versions of this interface were capable [3], but were not flexible enough to meet the demands of more advanced trajectory generation algorithms. With increased demands on rover motion control, fueled in part by the latest research in motion planning, it became necessary to extend the interface. The extension, presented herein, allows increasing the flexibility of using motion trajectories, as well as improving the code reusability aspects of the software. Similar interface development efforts have been done previously in CLARAty [4], in particular in the domains of rover navigation [5] and image acquisition [2]. This work builds on its predecessors, but necessarily differs in many ways due to the particulars of interfacing for motor control.

The topic of reusable application programming interfaces for robot actuator control has received a fair amount of attention. Most architectures emphasized some aspects of this topic over others. Functional aspects of robotics systems have been studied in [6]. A higher-level perspective on robot control was used in [7][8]. However, in this work we focused on all relevant facets of actuator control, from application programmer's experience using the interface to maintainability and efficiency of the code. Some recent work in other projects addresses the ease of interfaces and code reuse as well [9][10], however our approach differs in a number of features and may be a potential alternative. We hope that the ideas presented herein could serve to encourage the inquiry into good programming interfaces to robotics motion control.

2. Requirements of the Motion Control Interface

The design of the presented motion control interface addresses a number of key requirements. Many of these requirements are shared with other components of CLARAty, namely camera interfaces [2]. Though most of the requirements were eventually supported in the final design, some were more influential than others. In this section we discuss the driving requirements.

Requirement 1: Convenience of basic operations. One of the most important requirements of an effective motor API is the convenience and simplicity of performing common operations with the motors. This is especially important for projects with high turn-over rates of developers, such as research tasks at educational institutions. Simplicity of the interface is also helpful for maintenance and code reuse.

From our experience, the most frequent operations include moving the motor:

- to achieve a motion target (*setpoint*),
- to follow a specified trajectory profile.

These motions are specified in terms of (angular or linear) distance, velocity, acceleration or other quantities. This requirement states that the user shall be able both to specify the above motor motions and to execute them on the motor easily, e.g. using only a few lines of simple code.

Requirement 2: Accessibility of low-level functionality. Application developers are sometimes interested in having fine control over functionality of the hardware motors, as this is essential to making some algorithms work well. This requirement states that the user shall be able to manipulate the full set of features of the motor hardware using the interface. The user shall be assured that no other part of the interface will have any effect on the hardware, while the user maintains low-level access to it.

Requirement 3: Stability of the interface. A large variety of motor hardware is being used in robotics, and general standards for motor programming have not yet been developed. Figure 2 illustrates this variety with two particular research prototype rovers at JPL. The FIDO rover shown in the figure on the left features a centralized hardware architecture. This rover performs both trajectory generation and servo control functions in software on the main computer. The Rocky8 rover on the right uses dedicated microcontrollers to perform these motor control functions in hardware.

This requirement states that the motor API shall support the presently available motor hardware using

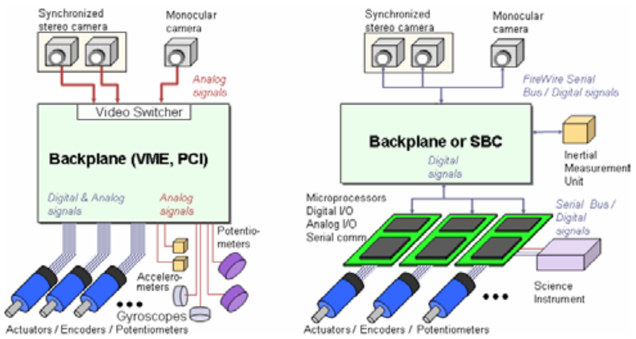

Figure 2. A comparison of motor control of two JPL research prototype rovers: FIDO (left) and Rocky8 (right). FIDO performs centralized motor control, while Rocky8 features a distributed system with dedicated controllers.

the same basic set of methods. It also shall be able to support the likely future variants of it. The interface shall be extendible to support future capability without major changes.

Requirement 4: Resource Sharing. Some applications require the motor hardware resources to be shared among different software processes. For example, several programs may require access to a particular motor. This requirement states that the motor API shall allow two or more separate software processes to access and utilize the motor hardware. It shall also implement error checking to inform the users if the hardware is unavailable due to resource sharing.

Requirement 5: Synchronization. Many algorithms require robot motions to be synchronized in time. This is particularly necessary for locomotion, advanced maneuvering and manipulator motion. This requirement states that the motor API shall allow synchronizing the operation of two or more hardware motors. The interface shall provide feedback to the user regarding the synchronization capability that is possible and the quality of synchronization during motion.

3. Motion Control Interface Design

In this section we describe our solution to the problem of designing an effective motion control interface. We will present the major design decisions and the features that result from it. We will also note the aspects of the design that satisfy the previously outlined requirements.

3.1. Motor Class Hierarchy

The first version of the motor API class hierarchy in CLARAty featured a base motor class and a derived class that specialized the motor to the hardware. While this design was very simple, it only allowed for a hardware adaptation, but not for a functional adaptation. For example, a joint could not be derived from a motor class in this design. To address this limitation, we introduced the *bridge* pattern [11] in the second version that separated the functional motor abstractions from the implementation abstractions. However, this resulted in a significant increase in complexity without adding significant flexibility. The complexity was caused by the development of parallel hierarchies for both motor hardware code and the functional motor abstractions. Both hierarchies became similarly complex, which effectively doubled the complexity of the design. On the other hand, the flexibility was still insufficient because adding new functionality required modifying both hardware code and motor abstractions.

The third version, presented here, combines the simplicity of the first version with some of the features of the second version in a simplified class hierarchy, shown in Figure 3. Similar to UML notation, arrows in the figure indicate inheritance, and diamonds indicate containment.

The *generic* device class defines a set of member functions that represents general functionality of all devices. The *base* class, derived from the generic class by simple inheritance, defines a set of member functions for performing the common operations of the motors. The application code can be written using the base class without specifying the motor hardware to be used.

The *adaptation* class is in turn derived from the base class. The principal purpose of this class is to map the logical motor operations of the base class to the physical motor operations of the hardware class (bottom right of the figure). The adaptation class can also be useful for storing user settings of the motor.

The *hardware* class represents the physical hardware, independent of how the architecture intends to use it. Hence, it is not constrained by class inheritance. Low-level operations of motor hardware can be implemented as member functions of the hardware class.

The above class hierarchy forms the foundation of the motor interface. The generic and base classes in the figure are unique in the API. The adaptation and hardware classes are duplicated and tailored to the motor hardware. Due to this arrangement, the interface is positioned to satisfy the Requirement 2 (accessibility of low-level functionality). By eliminating the bridge pattern, we traded some flexibility (i.e. limited the extendibility of the motor class) for simplicity and better maintainability.

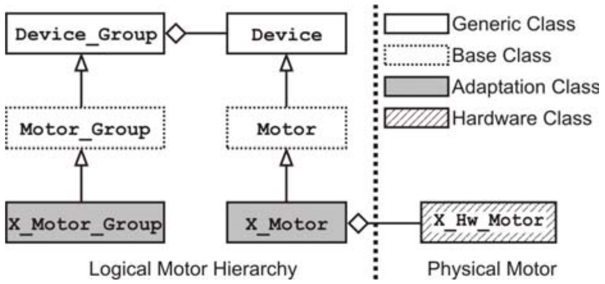

Figure 3. Class Hierarchy of the Motor Interface.

3.2. Motor Properties

The information stored in the motor class can be divided into two types: the properties of the motor itself and the properties of the motion that it must execute. Here we discuss the former, while the next section is dedicated to the latter.

Motor properties are further subdivided into the motor *model* and *parameters*. The model contains information that describes the hardware and therefore remains constant across applications. Common examples of the motor model information include limits on position, velocity and acceleration. Similar to general mechanism models [13], the motor model code and data can be reused to develop simulations of the motors. In our implementation of the motor interface, we dedicated a special class to handle the motor model information. This approach makes it convenient to store and to load the model from a file, as well as to share it between copies of the same motor adaptation. In developing the motor model class, we attempt to include the values that are relevant for most motors, so that this class can be reused when controlling various motor adaptations.

Motor parameters, on the other hand, are values that can be modified by the application. A common example of such parameters is controller gains. The meaning of these parameters is typically specific to a particular motor adaptation, which makes it difficult to create generic collections of motor parameters. Implementing particular parameters in motor adaptations is the approach we recommend.

3.3. Motor Commands

In this section we describe the second type of motor information, the properties of the motion command that the motor must execute. Developing a general approach to describing motor motions is a challenging task because it depends not only on the variety of motor hardware, but also on the variety of applications and use cases. Here we present the software framework that captures most common specifications of robot motion.

From a motor control point of view, the typical motor motion commands can be classified according to the hierarchy in Figure 4. At the highest level, a motor command is classified according to the control mode: *setpoint* or *trajectory*. Both modes are further classified according to the actual quantity being commanded: motor position, velocity, torque, etc.

The setpoint control mode provides the motor with an instantaneous target value that the motor will try to

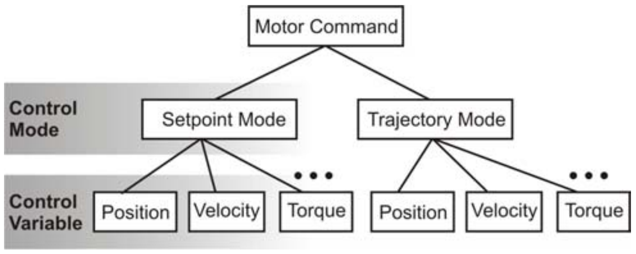

Figure 4. A hierarchy for classifying typical motor commands.

achieve using its control law (typically using a PID controller) without the generation of a trajectory profile. An example of this motor command is moving N revolutions as fast as possible. The motor command in this example would be classified as setpoint mode with control variable of position (i.e. angular position of the motor shaft). Optionally, limits on derivatives of the control variable can be specified, if they are supported by the motor hardware. In our example, by specifying a limit on the first derivative, we could obtain a velocity-bounded motion.

In contrast, trajectory control mode refers to a motion command that must follow a prescribed trajectory profile. The profile also may be represented in terms of any control variable supported by the motor. This control mode has a number of important applications including manipulator control, advanced robot locomotion, etc. Coordinated motion of several motors can be achieved by computing and executing the trajectories that satisfy the time synchronization requirements.

In order to implement the trajectory mode of execution, additional features are necessary in the motor interface. The methods to represent, generate and execute the trajectories are covered below.

3.3.1. Representing Trajectories. Our goal is to enable the motor API to handle *all* types of trajectories that are used in robotics applications. This is a challenging task, and in our experience the best approach to achieving that is to classify the trajectories into two categories:

- *simple*, where we assume a simplest motion that satisfies the specified boundary conditions (also referred to as a *trapezoidal trajectory*), and
- *complex*, where we allow the trajectory to have an arbitrary profile.

In our experience, supporting both trajectory types in the motor interface has satisfied all desired motor motions so far.

Before we look at both trajectory types in more detail, it would be helpful to develop some notation.

Given a certain control variable of the motor, y, a trajectory τ is typically defined as a mapping from time t to y, also denoted as $\tau : t \rightarrow y$ [12]. Moreover, since we are concerned with motion of physical motors, we can make certain assumptions about this mapping. In particular, this mapping is a *function* in mathematical terms, since a physical motor cannot have more than one value of its control variable y at the same time. When discussing trajectories as functions, e.g. $y = \tau(t)$, we will refer to t as the *argument*, and y as the *function value*. The argument typically is a value of time, but can be another quantity. The function value can be a value of any control variable, such as position, velocity, torque, etc.

As suggested above, the term simple trajectory denotes a trajectory that is only expected to satisfy boundary conditions and does not specify the profile of the trajectory itself. The representation of the simple trajectory includes only a few parameters:

- the *boundary conditions*:
    - the initial and final values of the function value (and its derivatives),
    - the initial and final value of the argument;
- the *derivative limits*: the allowed ranges of variation of the function value and its derivatives.

In other words, the boundary conditions specify the target that the motor must achieve after executing the trajectory. The derivative limits specify the allowed ranges of values that the function (e.g. position) and its derivatives (e.g. velocity) can take on. This can allow us to make sure that the velocity of the motor never exceeds a certain range. As a side note, the derivative limits can be viewed as the model of the system performing the action, similar to the way the motor model in Section 3.2 described the capabilities of the motor.

Figure 5 shows an example of a simple trajectory, where we would like a motor to move N revolutions within t_f amount of time. The boundary conditions are as follows:

- the initial value of the function y is 0, and its final value is N revolutions;
- the initial and final values of all derivatives of y are 0 (the motor begins and finishes the trajectory at rest);
- the initial value of the argument is 0 (always the case for relative motions), and its final value is t_f.

This representation does not specify whether velocity and other derivatives of the function remain constant during the steady-state part of the motion. However, it assumes that they do, based on practical considerations. Also note that in this example, the second derivative (acceleration) is discontinuous, hence the third derivative (jerk) is unbounded. If necessary, we can use the same representation to bound the third derivative and represent a smoother motion.

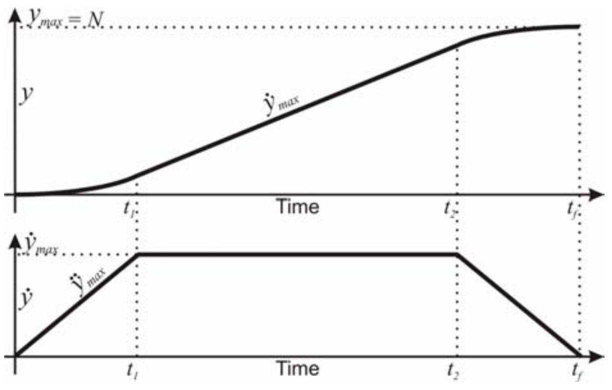

Figure 5. An example of a simple trajectory that achieves *N* motor revolutions in time *t_f*. Top: the trajectory in terms of angular position; bottom: the corresponding (trapezoidal) trajectory in terms of velocity.

As we have seen, a simple (trapezoidal) trajectory can be represented by only a few parameters. A complex trajectory, however, cannot benefit from this compactness, because it must prescribe the value of the function at every value of the argument. In order to represent the complex trajectories, we rely on a related software framework in CLARAty, referred to as *Math Function API*. Similar to the general motor interface that is the object of this paper, Math Function API is a general framework for representing and using any mathematical function.

The class hierarchy of the Math Function framework is shown in Figure 6. Math Function API supports common types of functions, including polynomial and trigonometric functions (e.g. Sine_Function and others), as well as sampled functions (a discrete set of values, capable of representing an arbitrary sampled function). The framework supports concatenating individual instances of Math Function classes into a piecewise function. The *composite* pattern [11] is utilized to ensure that an instance of the piecewise function class can be used as any other math function as well. The features of the Math Function classes include not only evaluating the functions they represent, but also computing their derivatives and integrals.

Thus, the task of representing the complex trajectories is handled by the Math Function framework, specifically designed to represent arbitrary functions. This relationship is well aligned with other approaches of code reuse in CLARAty.

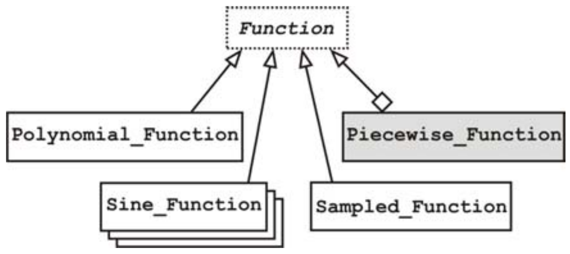

Figure 6. Class hierarchy of the Math Functions software framework, used for representing complex trajectories.

3.3.2. Generating Trajectories. Now that we have developed the methods for representing both types of motor trajectories, we will discuss how they can be generated. Trajectory generation refers to finding the complete specification of the trajectory, given its boundary conditions. In the case of the example of the trapezoidal trajectory in Section 3.3.1, trajectory generation would involve finding the times t_1 and t_2 of the transitions from ramp-up to plateau and from plateau to ramp-down, respectively. In case of the complex trajectory, the generation would involve computing all values that represent it. For example, in case of a piecewise polynomial function, this would include the coefficients of all the polynomials involved. In the general case of the sampled function, this would involve computing all of its samples.

The generation of most practical simple (trapezoidal) trajectories is straight-forward and can be done in closed-form. However, since the complex trajectories can be arbitrary, their generation can be arbitrarily difficult. For example, some trajectory generation algorithms could involve iterative gradient descent methods or parametric optimal control. In order to preserve the uniformity of the motion representation part of the motor interface in light of these differences, we decided to separate trajectory generation from the representation of trajectories themselves. Thus, trajectory generators are separate software elements in CLARAty, widely ranging in complexity. However, the communication between them and the motor interface occurs via the easy-to-use trajectory representations, described in Section 3.1.1. This approach allows the trajectories to be light-weight both in terms of storage and meaning. In turn, this facilitates learning and maintenance of the motor interface.

3.3.3. Executing Trajectories. As was mentioned in Section 2 and illustrated in Figure 2, there is a large variation in motor hardware in robotics. This variation is especially relevant for executing trajectories. In order to enable the motor interface to be general and re-usable for a variety of motor hardware, we must build generality into the method of executing trajectories. This is the essence of the Requirement 3, and our method achieves this requirement by placing the specifics of trajectory execution into the hardware-specific code, namely the adaptation and hardware classes in Figure 3. Thus, the application code is able to manage abstract representations of trajectories *only* and pass them directly to the hardware-specific code for execution.

Algorithm 1 is a code example that demonstrates the simplicity of setting up a motor and commanding its motion. The lines 1 and 2 setup the hardware motor class and its logical adaptation, respectively. Line 3 sets the motor in setpoint control mode, the default control variable is angular position. Line 4 specifies the motion by 1 radian, and line 5 enacts the motion. Separating setting up the motion and enacting it (lines 4 and 5) enables better error handling: the function start() executes only if there were no errors with commanding the motion. Line 6 blocks until motor finished executing the previous trajectory. The motor is set to the trajectory mode in line 7, and the following line provides the residual value of the constructor of Trapezoidal_Trajectory (containing the representation of a simple trajectory type) as the trajectory to follow. The single argument means the goal of angular distance 1.0, and the remaining parameters are set at defaults, obtained from the motor model. Finally, line 9 executes the trajectory. This example illustrates how our motor interface satisfies the Requirement 1. The simplicity of performing this typical motor operation was enabled by the motor class hierarchy of choice and crystallizing the particulars of motor control in a few intuitive parameters.

One of the typical differences in motor hardware is the method of enacting motor motion. Some motor controllers implement certain common trajectory types in hardware. Other motors, including the motors in the

```
1. X_Hw_Motor hw_motor(parameters)
2. X_Motor motor(hw_motor)
3. motor.set_control_mode(SETPOINT_CONTROL)
4. motor.set_setpoint(1.0)
5. motor.start()
6. motor.wait_until_done()
7. motor.set_control_mode(TRAJECTORY_CONTROL)
8. motor.set_trajectory(Trapezoidal_Trajectory(1.0))
9. motor.start()
```

Algorithm 1. Using the motor interface to setup a motor and command motions.

JPL FIDO rover, are controlled in software and cannot benefit from hardware-generated trajectories. Therefore, the motor adaptation classes must include the logic to translate the representations of trajectories from Section 3.3.1 into executable motor commands. In the case of software motors, this would include invoking the trajectory generation code to compute trajectory parameters in software, as was alluded to in Section 3.3.2.

3.4. Motor Groups

In order to enable synchronization of motors and to satisfy the Requirement 5, we propose a special class hierarchy, Motor_Group (Figure 3, left), that parallels the motor hierarchy (Figure 3, center). While the Motor classes represent a single motor device, the Motor_Group represents an arbitrary number of motors. It also contains additional functionality to command and execute the motion of its motors in a coordinated fashion. The Motor_Group inherits from the Device_Group class due to many similarities between the motors and other devices in CLARAty, including cameras [2].

The Motor_Group inherits member functions to append and remove Motors from the container. The template function, `for_each`, supplies a way for some member functions of Motor to be called for every motor in the group. This allows, for example, setting control mode of each motor to the same value in one line of code. The Motor_Group class also supplies new member functions that allow synchronized motions by all motors in the group. Algorithm 2 demonstrates how a group of motors can be used for coordinated motion. Lines 1-5 set up the motor group, consisting of two motors. Line 6 uses the `for_each` method to set all motors in the group to trajectory control mode. Line 7 sets up trajectories for each motor in the group, where the order of arguments corresponds to the order of appending motors to the group. The motor group automatically ensures that the motors begin and end motion at the same time, as fast as motors allow. Line 8 initiates the motion.

```
1. X_Hw_Motor hw_motor1(parameters)
2. X_Hw_Motor hw_motor2(parameters)
3. X_Motor motor1(hw_motor1)
4. X_Motor motor2(hw_motor2)
5. X_Motor_Group motor_grp(motor1, motor2)
6. motor_grp.for_each(set_control_mode,
                               TRAJECTORY_CONTROL)
7. motor_grp.set_trajectory(Trapezoidal_Trajectory(1.0),
                              Trapezoidal_Trajectory(2.0))
8. motor_grp.start()
```

Algorithm 2. Using the motor interface to setup a motor group for coordinated motions.

Thus, common operations with motor groups are greatly simplified by the motor interface. Commanding synchronized motions is nearly as straight-forward as commanding individual motors. The proposed API framework allows intuitive naming and structure of the relevant interface commands.

3.5. Motor Resource Sharing

Enabling sharing the motor resources between non-cooperating software processes (satisfying the Requirement 4) is one of the most challenging aspects of designing the motion control interface. As suggested above, there are two general types of operations in using the motors:

- setting and querying motor parameters and motion,
- commanding the specified motion to start.

The problems due to resource sharing, including deadlock and starvation, are caused by errors in providing access to processes to perform either action. These problems are addressed by separating logical and physical motors, as shown in Figure 3. Further, we establish a rule that no more than one physical motor object must exist for any motor in the system. This object is shared among all threads.

Similar to the CLARAty camera interface [2], a hardware motor class is responsible for getting/setting the parameters of the physical device, commanding and executing a motion. It also assists in implementing a locking mechanism which allows a logical motor to block any other logical motor from setting motor parameters or commanding a motion.

In contrast, a logical motor represents the user's view of the motor. Multiple instances of the logical motor may exist in the system. Parameter settings made in one logical motor are cached locally and do not affect the state of another logical motor, even if both refer to the same piece of motor hardware. The member function of the logical motor that starts execution of a motion affects an atomic operation which both sets the physical motor parameters to match the user's view of the motor state and starts executing the motion. Using this interface, the user does not need to write any special code to maintain task safety: thanks to the motor interface, the code written for single-threaded operation will operate correctly if the motor resource is shared between multiple threads [2] as long as the motor adaptation is implemented in a thread safe manner.

4. Conclusions and Future Work

The presented motion control interface was implemented and tested on research prototype rovers at JPL. Figure 7 shows an example of the FIDO rover executing a maneuver amid dense obstacles using the motor interface. The interface made it easy to program the coordinated motion of 12 steering and drive motors of the rover's mobility system, as well as the precision motions of pan and tilt joints in its mast.

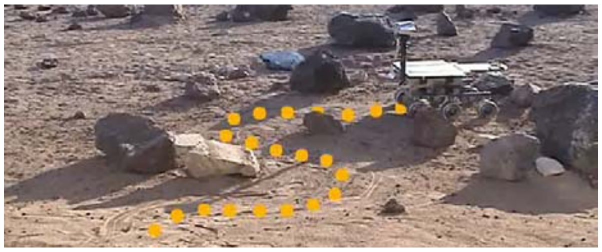

Figure 7. FIDO rover executes a maneuver using the motor interface.

We presented an approach to developing a motion control API that maximizes the effectiveness of learning, usage and maintenance. This interface is likely to be helpful for applications that feature high developer turn-over rates and pursue code reuse. In future work, we plan to continue extending this interface to new robotics platforms intended for both space and terrestrial robotics research.

5. Acknowledgments

We thank Rich Petras, Won Kim, Khaled Ali, Lorenzo Flueckiger and many others for the helpful ideas they contributed. We would also like to thank Mike McHenry, Jeffrey Edlund, Daniel Gaines, Tara Estlin, Daniel Clouse, Robert Steele, Hanz Utz and others for participating in the code review process. Our thanks also go to our sponsor, the NASA Mars Technology Program, for their support. The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology with contributions from NASA Ames Research Center and Carnegie Mellon under a contract to the National Aeronautics and Space Administration.

6. References

[1] I. A. Nesnas et al., "CLARAty: Challenges and Steps Toward Reusable Robotic Software", *International J. of Advanced Robotic Systems*, 3(1), pp. 23-30, 2006.

[2] D.S. Clouse, Issa A.D. Nesnas and C. Kunz, "A Reusable Camera Interface for Rovers", Int. Conf. on Robotics and Automation Workshop – Software development and integration in robotics: understanding robot software architectures, Rome, Italy, 2007.

[3] R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, H. Das, "CLARAty: Coupled Layer Architecture for Robotic Autonomy", JPL Technical Report D-19975, Dec 2000.

[4] I. A. Nesnas, "The CLARAty Project: Coping with Hardware and Software Heterogeneity", *Software Engineering for Experimental Robotics*, *Springer Tracts on Advanced Robotics*, ed. D. Brugali, 2006.

[5] C. Urmson, R. Simmons and I. Nesnas, "A Generic Framework for Robotic Navigation", in *Proc. of the IEEE Aerospace Conference*, Big Sky Montana, 2003.

[6] G. Pardo-Castellote et al, "Controlshell: A Software Architecture for Complex Electromechanical Systems", *Int. Journal of Robotics Research*, 17(4), 1988.

[7] J. Borrelly et al, "The ORCCAD Architecture", *Int. Journal of Robotics Research*, 17(4), April 1998.

[8] R. Simmons and D. Apfelbaum, "A Task Description Language for Robot Control," in *Proc. of the IEEE/RSJ Intelligent Robotics and Systems Conf.*, 1998.

[9] The Orca Robotics Project, http://orca-robotics. sourceforge.net.

[10] R.T. Vaughan, B. Gerkey and A. Howard, "On Device Abstractions For Portable, Resuable Robot Code", in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robot Systems*, pp. 2121-2427, 2003.

[11] E. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass: Addison-Wesley, 1995.

[12] S.M. LaValle, *Planning Algorithms*, Cambridge University Press, 2006.

[13] H.D. Nayar, I.A. Nesnas, "Re-usable Kinematic Models and Algorithms for Manipulators and Vehicles", in *Proc. of the Int. Conf. on Intel. Robots & Systems*, 2007.