Abstract

A Reusable Operational Software Architecture for Advanced Robotics

Robotic software can be broadly categorized into two major levels. The first is the actuator control software. The second level is system control software. There are three layers within the system control software level. The top-most is the man-machine interface. The lowest is the real-time control layer. The middle layer is known as the operational software layer.

Current industrial robots are monolithic six degrees-of-freedom (DOF) manipulators that have minimal operational software requirements. On the other hand, advanced robots are based on modularity, redundancy, fault-tolerance, condition-based maintenance, and performance. The operational software layer for these robots should be general and reconfigurable, and should support kinematics, dynamics, deflection modeling, performance criteria, fault-tolerance, and condition-based maintenance. **This paper discusses the development of a reusable and general architecture for the operational software layer**. This development includes:

- *requirements generation for an advanced robotic software system,*
- *the selection of a development, execution, and test environment,*
- *the development of a reusable software architecture to support Operational Software Components for Advanced Robots (OSCAR),*
- *the development of applications to demonstrate OSCAR in simulation on a wide variety of robots and in real-time on a seventeen degrees-of-freedom dual-arm manipulator,*
- *the design and development of experiments to validate the advantages of OSCAR,*

The development of OSCAR was based on object-oriented design which involves the development of software components that are extensible and have standardized interfaces. The application of this philosophy led to the break-down of the advanced robotics domain into sub-domains. An analysis of these sub-domains led to the identification of software components that made up the each sub-domain. A detailed analysis and design of these components then followed.

The OSCAR environment supported distributed control and was equally applicable to simulation and real-time control. OSCAR was demonstrated in simulation using 3D graphics on a wide variety of robots. A 17-DOF dual-arm manipulator (by Robotics Research Corporation) coupled with various manual controllers provided the real-time test-bed for OSCAR. Generalized criteria-based kinematics and fault-tolerance were demonstrated on the 17-DOF manipulator. OSCAR also provided the building blocks for generalized software architecture for manual controller integration.

Applications and experimentation validated the effectiveness of OSCAR. Achieved goals of this software architecture were demonstrated in terms of a series of applications. The experiments demonstrated that OSCAR provided a 30% improvement in its 'ease of use' and an approximately 200% reduction in program development time as compared to the other robotic software in use at the Robotics Research Group at the University of Texas at Austin.

Overview

A Reusable Operational Software Architecture for Advanced Robotics

Introduction

Robotics research today is focusing on developing systems that exhibit modularity, flexibility, redundancy, and fault-tolerance [1]. The development of this functionality puts an extra burden on the software control system that has to not only exhibit modularity and scalability, but has to support a generalized software architecture that applies to all classes of intelligent reconfigurable machines (robots). This architecture must support reuse and should be built-up on a foundation of generalized kinematics, dynamics, deflection modeling, performance criteria, criteria-fusion, condition-based maintenance, and fault-tolerance. These are the essential characteristics of an advanced robotic system and are discussed in [1-5].

A robotic software system at the system controller level can be broadly divided into three layers. The lower-most layer of this system is the hardware-interface layer. The purpose of this layer is to interface with the servo control software, peripheral devices, and communication buses. Real-time constraints on this layer are the most stringent because it interacts with the external hardware. A parallel software execution environment is best suited for this layer. This is because the software in this layer has to handle multiple asynchronous events (both external and internal) in real-time (less than 1 millisecond).

The top-most layer of a robotic software system is the robot programming language. This layer provides the man-machine interface for human intervention. In this layer, programs written in the robot programming language are converted into appropriate commands for the middle layer. These commands are then translated into actuator position, velocity and torque commands by the middle layer that are then sent to the lowest layer.

The middle-layer of the robotic software system is the focus of this paper and is referred to as the operational software layer. Most industrial robots are six degrees-of-freedom (DOF) arms and have geometries that allow for simple kinematics and dynamics. As such, their operational software layer is not very demanding and offers limited capability. Advanced robots, which exhibit reconfigurability, redundancy, and fault-tolerance, place exceptional demands on the operational software. It is desired that this layer of software be robot independent, reconfigurable (if there is a fault), and support software reuse and rapid-prototyping. Also, the operational software architecture should support generalized kinematics, generalized dynamics, deflection modeling, performance criteria, and fault-tolerance in a reusable and modular software architecture

Literature Review

There are two predominant software design methodologies. These are *structured design* and *object-oriented design* (OOD). Structured design is based on data-flow, where data flows from one subroutine (or procedure) to another, thereby undergoing transformations. In this design philosophy, data and instructions are kept separate. OOD allows the building of software as components with standardized interfaces and reuse capability. Reuse is a result of the generality and extensibility of these components. Extensibility is achieved in two ways. Specialization allows for the customization of an existing component by extending, constraining, or even changing the object without modifying the already existing code. This is also called *inheritance*. The second means of extensibility is through *containment*. In this, a set of components can be combined together to form a new component. Over structured design, OOD offers the advantages of modularity, reusability, and standardized interfaces. [6]

Various robotic software architectures have been developed at all levels of robot control. The majority of these architectures have concentrated on the man-machine interface (top layer) and the hardware-interfacing layer (bottom layer). RCCL was a ‘C’ programming language based robot control environment that used UNIX as the operating system. Again, the operational layer was minimal in RCCL as it primarily supported standard industrial robots [7]. RIPE is a Robot-Independent Programming Environment that used object-oriented design to provide a standard programming interface for robots [8]. GISC extended RIPE to distributed computing, but did not address the operational software needs for redundant systems [9]. The SMART architecture addressed robot kinematics and dynamics. However, it did not address multi-criteria redundancy resolution. Additionally, SMART is based on procedural design limiting its extensibility [10]. The C^H programming language developed by Cheng [11] supports data types for robotic computations. However object-oriented programming support is lacking. Functionality provided in C^H is either available or can be easily made available in C++. Commercially, Control Shell is a component-based real-time programming environment that uses object-oriented design. A graphical environment is provided for component assembly. Examples of components supported are PID control, impedance control, trajectory control, etc. Control shell has a minimal operational software layer but provides an efficient environment into which OSCAR could be integrated [12].

Architecture Requirements

A software architecture is defined by the demands placed on it. Therefore, it is essential to have a firm understanding of what these demands are. The design requirements for this effort are summarized below:

- **Open System:**
  An open system is one that allows user modification, extensibility, and integration with other systems.
- **Reusable:**
  The software must have a clear interface that gives the components a black-box ‘look-and-feel.’ Additionally, the user should be able to selectively modify, add, or constrain the functionality of a component.
- **General:**
  The software architecture should make no assumptions that limit the future integration of any conceivable manipulator.
- **Applicability to Real-time Control and Simulation:**
  As most concepts supported by this architecture are under development, the architecture should be equally applicable to simulation and real-time control. This feature allows for algorithm testing in simulation before use on physical hardware. The necessity for compatibility with simulation is even greater as the operational software architecture being proposed in this research allows for extensibility and user-modification. The user of this architecture should be able to test any extensions or modifications to the software before controlling a physical robot.
- **Reduce Program Development Time:**
  The design requirements mentioned above should lead to a 50% reduction in program development time as compared with advanced robotic software previously used at University of Texas Robotics Research Group (UTRRG).

System Development

Along with the design of the OSCAR architecture, it was important to put in place a software development, execution and test environment. The software development environment would focus on the operating systems, the programming language, and various development tools. The execution environment focused on the execution operating system, the physical machines that were to be controlled, and the computer hardware. The test environment involved the use of a simulation environment for program testing. The final system selected for this research used 'C++' as the programming language, VxWorks was the real-time operating system, UNIX was used for execution of software in simulation mode, Silicon Graphics computers were used for 3D graphical simulation, and peripheral hardware such as manual controllers were used for application development. A 17 DOF Dual-Arm Robot (from Robotics Research Corp.) was used for demonstration and testing of the software. This environment is depicted in Figure 1.0. [13]

Architecture Development

There are three major steps involved in designing object-oriented software components. The first step is the analysis of the problem domain. A set of entities and the relationship between them are laid out in this phase. The next phase is the software design phase. Decisions are made based on execution environment, programming language, and operational constraints. Entities identified during the analysis phase map into classes and objects in the design phase. The third step is the implementation of the software. This includes filling in the details of class data structures, adding internal functionality to support overall class functionality, writing member functions, and unit testing. These three steps are generally iterative [6]. The next sections discuss the application of OOD to the advanced robotics domain.

Analysis

As discussed previously, the advanced robotics domain encompassed generalized kinematics, dynamics, deflection modeling, performance criteria, fault-tolerance, and condition-based maintenance. The areas constituted the sub-domains of the advanced robotics domain. An analysis was performed on each of these sub-domains to identify the key components comprising them. Also, domains necessary to support the development and testing of this architecture were identified.

Domain	**Functionality**
Mathematical	• Support for linear algebra constructs • Matrices, vectors, tensors, transformations, Euler-angle representations, etc. • Provides building blocks for operational software constructs
Robot Data	• Support for storing robotics-related data and perform error checking on the data • Automatic initialization from data files • Examples are DH parameters, inertia tensors, center-of-gravity data, etc.
Utility	• Basic components to aid program development • String classes, Timing classes, Error handling
Others	• Key data structures such as linked-lists, trees, etc. • Commercially available container components • Wrapper classes for encapsulating operating system functionality
Forward Kinematics	• Position, Velocity, and Acceleration level • Kinematic influence coefficients [3]
Inverse Kinematics	• Resolved-rate motion control [14] • Direct-search technique [2] • Hybrid inverse technique [13] • Closed-form and generalized approaches [15] • Level III Fault-tolerance [16]
Performance Criteria	• Abstract components for performance criteria • Example performance criteria [5] • Criteria fusion [2]

	• Blackboard framework for information exchange [6]
Dynamics	• Newton-Euler inverse dynamics [16] • Lagrangian dynamics [3] • Complete description of physical plant
Deflection Modeling	• Deflection prediction in serial manipulators [4] • Deflection due to joint and link compliance • Generalized spring
Input Output Devices	• Abstract input-output device classes • Serial-ports, Keyboard, memory-mapped IO, device drivers
Communication	• Inter-process communication • Network communications • Client-server classes using TCP-IP

Table 1.0: Key Domains of OSCAR.

A summary of all the sub-domains and their functionality is shown in Table 1.0. The two shaded blocks in Table 1.0 shows the supporting and the testing domains, respectively. The clear block of Table 1.0 shows the key robotic domains. The reader should refer to [13] for a detailed analysis of each domain.

Design and Implementation

The design issues are not significant at the domain level. However, design does achieve a significant role at the component level. That is, whenever an analysis on a particular component is completed, design plays a key role in the generation of implementation level specifications. Some of the issues for component design are outlined below:

- All class names were prefixed with the letters 'RR' to avoid name space pollution.
- As real-time control was a requirement of this architecture, dynamic binding and operator overloading were judiciously used.
- Wherever possible, components supported a dual-interface. One tuned for computational efficiency and the other for ease-of-use.

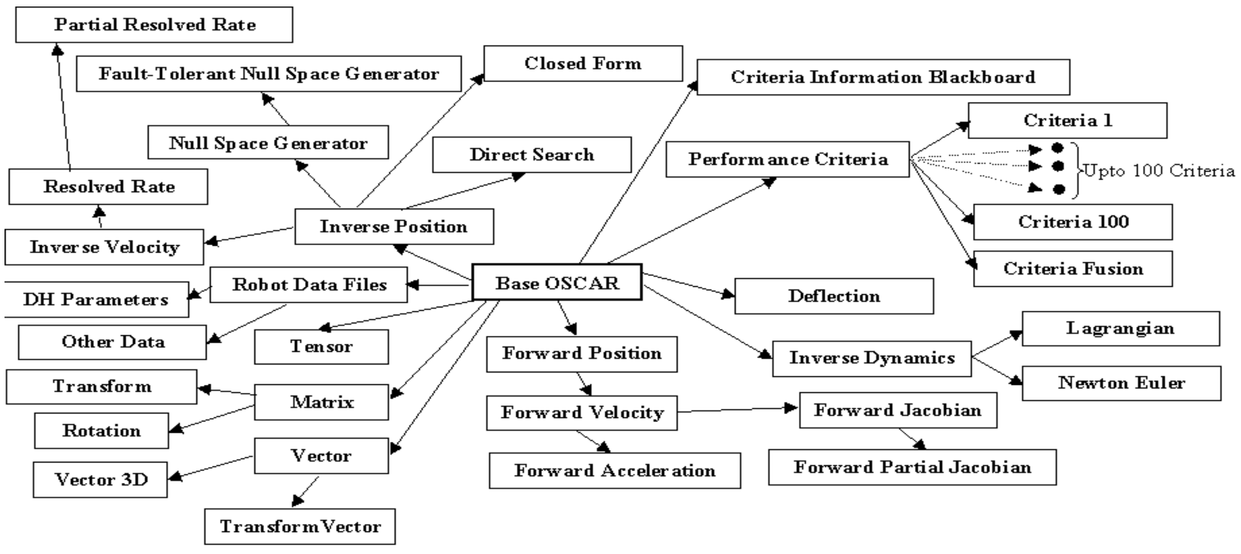

Figure 2.0: High-level Class Hierarchy Specification.

The reader should refer to [13] for the detail design of each component in the OSCAR architecture. Figure 2.0 shows a high-level description of the class/domain hierarchy (which implies a large population of components). The arrow indicates inheritance and points towards the child.

Example Application

An example of program development using OSCAR is shown below. This sample shows the computation of the inverse position solution for a manipulator whose DH parameters are described in the "puma.dh" file. First a forward kinematics object (*RRFKPosition*) is created to compute the initial end-effector location based upon the initial joint configuration. Then, supplying the initial joint configuration and the maximum desired error creates an inverse position (*RRIKJForSix*) object. The inverse object is then repeatedly invoked to evaluate the inverse position solution for a changing end-effector location.

```
RRFKPosition fowkin("puma.dh");                          // create a forward position object
RRVector3 toolPoint;                                      // create a vector to hold the tool point
RRVector initialJoints(fowkin.GetDOF()));                 // create a vector for holding initial joint configuration
RRVector jointSolution(fowkin.GetDOF()));                 // create a vector for holding the inverse solution
toolPoint[2] = 13.0;                                      // set the tool point to be 13 units in the 'Z' direction of the link
frame of the last joint.
RRXform hand;                                             // create a 4X4 matrix to hold the end-effector pose
fowkin.SetToolPoint(toolPoint);                           // inform 'fowkin' of the tool point location
hand = *fowkin.GetHandPose(initialJoints);               // compute the starting hand location
RRIKJForSix invkin("puma.dh", initialJoint, 0.1);         // create an inverse kinematics object
invkin.SetToolPoint(toolPoint);                           // inform 'invkin' of the tool point location
for(int I = 0; I < 30; I++){                             // increment the 'X' coordinate of the
    hand.at(0,3) += 0.1;                                 // current hand location by 0.1 30 times
     if(invkin.GetJointPosition(hand, jointSolution) > 0)   // and compute the inverse position
         cout << jointSolution * RADTODEG << endl;       // solution and print it
   else cout << "singularity" << endl;                   // if no solution, print "singularity"
}
```

Experiments and Conclusions

One of the research objectives of OSCAR was to cut program development time by 50%. Experiments were conducted with OSCAR to test this hypothesis. These experiments involved eight experimenters and ten experiments in increasing order of complexity. The experiments were conducted over a period of four months. The weight of each experimenter (computed based on aptitude, motivation, experience) was used for normalization.

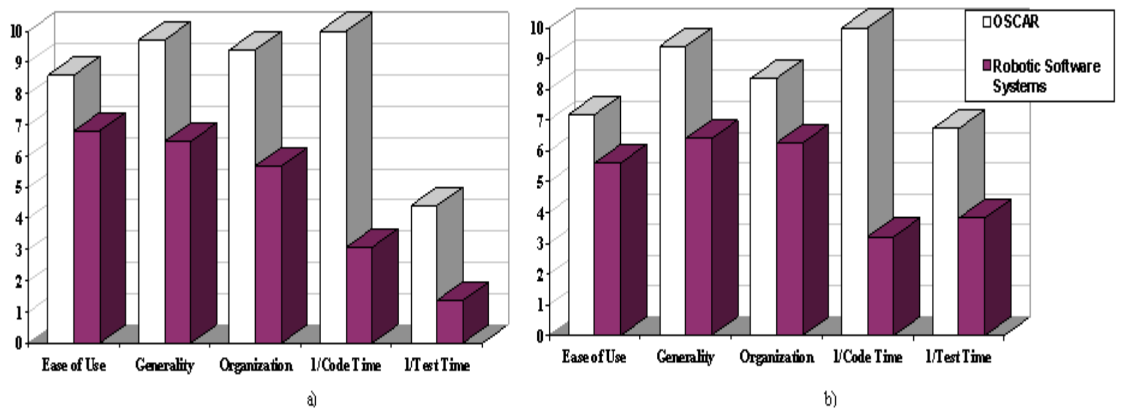

Figure 3.0: OSCAR Comparison. a) Mathematical Software, b) C/C++ Software.

The net result of the experiment led to a comparison of OSCAR with two types of robotic software environments. These were, Mathematical software (for example, Matlab and Mathematica), and C/C++ robotic software that was previously used at UTRRG. From Figure 3.0 it is clear that OSCAR shows improvement in all aspects it was evaluated in. Current estimates put OSCAR at 27,000 lines of C++ code, 60 classes and 550 methods.

References

1. Butler, M., and Tesar, D., "A Generalized Modular Architecture for Robot Structures," *Manufacturing Review*, Vol. 2, No. 2, June 1989, pp. 91-118.
2. Hooper, R. N., and Tesar, D., "Multicriteria Inverse Kinematics for General Serial Robots," Ph.D. Dissertation, The University of Texas at Austin, 1994.
3. Thomas, M., and Tesar, D., "Dynamic Modeling of Serial Manipulator Arms," *ASME Journal of Dynamic Systems, Meas. and Control*, Vol. 104, No. 3, 1982, pp. 218-228.
4. Fresonke, D.A., Hernandez, E., and Tesar, D., "Deflection Prediction for Serial Manipulators," *Proc. IEEE Conf. on Robotics and Auto.*, April 24-29, 1988, pp. 482-487.
5. Van Doren, M., and Tesar, D., "Criteria Development to Support Decision-Making Software for Modular, Reconfigurable Robotic Manipulators," Master's Thesis, The University of Texas at Austin, 1992.
6. Booch, G. 1994. *Object Oriented Analysis and Design with Applications*. Second Edition, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc.
7. Hayward, V., and Paul, R.P., "Robot Manipulator Control Under UNIX RCCL: A Robot Control "C" Library," *The Int. Journal of Robotics Res.*, Vol. 5, No. 4, 1986, pp. 94-111.
8. Miller, D. J., and Lennox, R. C. February 1991. "An Object-Oriented Environment for Robot System Architectures." IEEE *Control Systems*, pp. 14-23.
9. Burchard, R.L., and Feddema, J.T., "Generic Robotic and Motion Control API Based on GISC-Kit Technology and CORBA Communications," *Proceedings IEEE International Conference on Robotics and Automation*, April 1996, pp. 712-717.
10. Anderson, R.J. May 1993. "SMART: A Modular Architecture for Robots and Teleoperation." *Proc. of the IEEE Conf. on Robotics and Automation*, Atlanta, Georgia.
11. Cheng, H. H., "Extending C and FORTRAN for Design Automation," *ASME Transactions, Journal of Mechanical Design*, Vol. 117, No. 3, Sept. 1995, pp. 390-395.
12. Schneider, S.A., Chen, V.W., Pardo-Castellote, G., "Control Shell: A Real-Time Software Framework," AIAA *Conference on Intelligent Robots in Field, Factory, Service, and Space*, March 1994.
13. Kapoor C., and Tesar, D., "A Reusable Operational Software Architecture for Advanced Robotics," Ph.D. Dissertation, The University of Texas at Austin, 1996.
14. Nakamura, Y., *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley Publishing Company, 1991, pp. 336.
15. Sreevijayan, D., and Tesar, D., "On the Design of Fault-tolerant Robotic Manipulator Systems," *Proc. of the Third Int. Symp. on Meas. and Control in Rob.*, Turin, Italy, 1993.
16. Craig, J.J., *Introduction to Robotics Mechanics and Controls*. 1986, Addison-Wesley.