Proceedings of the 2006 IEEE International Conference on Robotics and Automation
Orlando, Florida - May 2006

Software Components and Frameworks for Medical Robot Control

Ankur Kapoor, Anton Deguet and Peter Kazanzides
Dept. of Computer Science
Johns Hopkins University
{kapoor, anton, pkaz}@cs.jhu.edu

***Abstract*— Robots are increasingly being used in Computer Integrated Surgery (CIS) systems, yet to our knowledge, there is no open source software that is specifically targeted at this application domain. In this paper, we derive unique requirements for medical robot controllers that are based on our experiences in the field. We present an overview of the second-generation open source software package (the *cisst package*), currently under development, that will include a family of application frameworks with dynamically loaded software components. We then focus on some some key design details, which include extensive application of the command pattern as well as a dynamic interface query mechanism, and present a simple example to illustrate the concepts.**

I. Introduction

In Computer Integrated Surgery (CIS), the computer integrates devices such as medical imaging systems, motion trackers and active or passive mechanical assistants (robots) to address a clinical need. CIS researchers must have access to hardware and software components for these types of devices. Although many medical robots have been developed for research, to our knowledge none of them utilize open architecture, open source robot control software. As a result, each group doing research in medical robotics (including our own) must create their own robot control software in addition to the hardware and software required by their application. The effort required to create this software is significant and a barrier to clinical deployment of new medical robot applications.

We are currently developing a second-generation software package (the *cisst package*), where we plan to include the components necessary for creating well-tested, clinically certifiable medical robot controller frameworks. This paper presents the high-level design requirements and an overview of the design, followed by a detailed discussion of specific design decisions, including a comparison to other approaches.

A. High-Level Design Requirements

Figure 1 shows a simplified architecture of a robot controller, which includes separate tasks for servo control (e.g., motor control) and supervisory control (e.g., trajectory control). Our overall goal is to provide software that enables a researcher to implement this architecture, but our design is heavily influenced by the following four distinctive requirements:

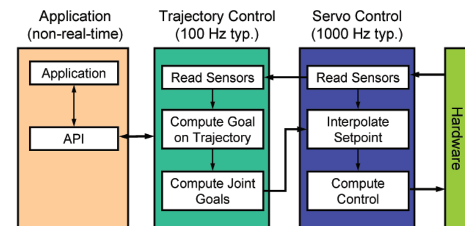

Fig. 1. Simplified Robot Controller Architecture

REQ-1: We must support the integration of both commerical off-the-shelf (COTS) and custom hardware/software for the basic functions shown in Fig. 1.

REQ-2: We must provide a documented and validated "core" software that facilitates research and clinical use.

REQ-3: The robot controller must serve as a component within a larger medical device and provide an interface that is consistent with, and possibly interoperable with, other types of components.

REQ-4: Our design should not constrain the different types of devices into a strict hierarchical relationship.

The first requirement reflects our experiences building different medical robots. For simple requirements, such as positioning tasks, we generally purchase COTS motion controllers and interface to them via the PC bus (ISA or PCI) or via a high-speed network (e.g., Ethernet, USB) [1]. These motion controllers contain software that provides low-level functionality such as motor control and coordinated motion in joint coordinates. In these cases, the researcher usually prefers a conventional operating system, such as Windows or Linux, to faciliate integration with other software components. At the other extreme, medical robots with novel mechanical designs and/or demanding performance requirements often require innovative algorithms at all software levels, including motor control, and may even necessitate the development of custom electronics[2]. For these systems, it is usually necessary to use a real-time operating system for (at least) the motor control software. Some projects lie in between these two extremes of development effort, such as a robot with a sensor-based control loop (e.g., visual or force feedback) that is implemented around a COTS motor controller[3].

The second requirement reflects our application domain, which is medical robotics. In most cases, our ultimate goal

0-7803-9505-0/06/$20.00 ©2006 IEEE

is to use the robot system clinically, which requires regulatory scrutiny and/or hospital oversight. Although it is the researcher's responsibility to design a safe system and to obtain the necessary certifications, we aim to provide high quality software and supporting documentation to facilitate this process. In addition, we believe that a clear separation of the "core" software from the application-specific software can dramatically simplify the learning curve for researchers that wish to use this software.

The third requirement states that a robot is only one type of component that can be used in a medical device. Therefore, our robot control software must exist within a larger application domain. To facilitate development, the robot software should present an interface that is consistent with other software-based devices. This is even more important when one considers that there are other technologies that provide some of the capabilities of robots (and vice versa). For example, the position of a surgical instrument can be reported by a robot system (assuming the instrument is attached to the robot) or by a tracking system (if the instrument contains the appropriate features, such as optical or magnetic markers). A researcher may initially design an application using a tracking system and later decide to switch to a robot to gain the benefits imparted by its motion capabilities. Our goal is to enable this exchange of similar devices without requiring the application software to be rewritten.

The fourth requirement reflects the complexity of the universe of devices that can exist. As a simple example, assume that a robot can contain position sensors (P), velocity sensors (V) and/or force sensors (F). There are 7 different device configurations that utilize at least one of these sensor types: P, V, F, P+V, P+F, V+F, P+V+F. These configurations cannot be represented by a strict hierarchy.

These requirements motivated us to develop software components and frameworks and to employ dynamic loading, which are first defined in the following section.

B. *Definitions*

There is no universally accepted definition of *software component* and it has been argued that attempting to create a classical definition (i.e., consisting of necessary and sufficient conditions) would be futile and harmful [4]. One point of contention is whether a software component must be in binary form [5] or whether it can consist of source code. Historically, the term software component was first applied to reusable elements of source code, but currently the most widely accepted definition requires a binary (or executable) form. In this paper, we use the terms *source code software component* (or *source code component*) and *binary software component* (or *binary component*) to clearly indicate the nature of each software component.

There are several definitions of an *application framework*. One simple definition is that "a framework is the skeleton of an application that can be customized by an application developer" [6]. It is also illustrative to focus on the control paradigm. With most reuse technologies the flow of control is managed by the user's software, which can, for example, instantiate and invoke several library classes. In contrast, a framework exhibits "inversion of control," which means that the framework calls the user-supplied software. This is sometimes called the *Hollywood Principle* – "don't call us, we'll call you". A well-known example of a framework is the Microsoft Foundation Classes (MFC).

There are two common uses of the term *dynamic loading*. The first occurs when the operating system must load a shared library at runtime to resolve a program's external references. The program cannot begin execution if one of the shared libraries is missing. The second use of this term occurs when a program loads a "plug-in" during operation using, for example, the `dload` or `LoadLibrary` functions provided on Linux and Windows platforms, respectively [7]. In this paper, we refer to this latter type of dynamic loading.

II. System Overview

Our goal is to create libraries of *source code components* that can be packaged into a family of *application frameworks* (executables) that correspond to the different levels of COTS hardware/software integration. The *application frameworks* will dynamically load "plug-in" *binary components* for hardware-specific and application-specific functionality. The entire *application framework*, with dynamically-loaded *binary components*, will itself serve as a *binary component* within a larger medical device.

A. *Source Code Software Components*

The *cisst package* consists of several software libraries that are grouped into the categories of *Foundation Libraries* (common tools, vectors, matrices, transformations, numerical methods, interactive environment), *Real Time Support* (operating system abstraction, device interfaces, tasks) and *Interventional Devices* (trackers and robots).

B. *Family of Application Frameworks*

As noted earlier, we have constructed medical robots with varying levels of reliance on COTS hardware and software. At one extreme, we use COTS hardware and software to provide the motor control and supervisory control tasks shown in Fig. 1. At the other extreme, we create custom software for all of these tasks. In the middle, we use COTS hardware and software for just the motor control task. This leads to the desire for the following *application frameworks* to implement the different architectures:

- **Complete Software Control:** This framework provides an application programming interface (API) and two periodic control threads: one for supervisory control one for low-level servo control. The framework allows the researcher to load new trajectory and servo control functions, as well as to provide the hardware interface (to simple I/O boards).
- **Software Supervisory Control:** The framework provides an API and one periodic control thread for supervisory control. The low-level servo control

is provided by an external device (e.g., a COTS motion controller board or custom hardware with embedded software).

Software API Only: The framework is primarily an API. There may be a thread to manage communications with the external device that provides the supervisory and servo control or it may be a library wrapper.

C. Related Open Source Software

Our design is most closely related to the Orocos project[8], although Orocos does not address our distinctive requirements (see Section I-A). In particular, Orocos aims to provide a complete software solution for robot control and is not designed to integrate with COTS hardware/software and is consequently not supported on the Windows operating system. Orca [9] is framework of binary components designed for mobile robots and is therefore quite different from our system. Similarly, the Modular Controller Architecture (MCA2) [10] was designed for mobile robots and consists of components with "sense" and "control" interfaces. The architecture for CIS systems recently proposed in [11] addresses requirement REQ-3, but it is unclear whether this will be available as open source software.

III. Design Features

The four high-level design requirements presented earlier led to some notable design features that are summarized here and presented in detail in subsequent sections.

From **REQ-1**, we conclude that we must be able to interchange software threads and devices to deal with the universe of intelligent and non-intelligent hardware. This is what motivates our design of Task and Device, described in Section III-A.

From **REQ-1**, **REQ-3**, and **REQ-4**, we conclude that we must interface with a wide variety of devices and tasks whose capabilities do not exist in a strict hierarchy. This leads to the discussion of the dynamic interface query mechanism, `Provides` function and composite devices in Section III-B. These requirements also imply that our frameworks must be flexible enough to adapt to the capabilities of the specific device being used, while still taking advantage of extra capabilities. This is also discussed in Section III-B.

REQ-2 implies that we must dynamically load hardware-specific and application-specific binary components into our "core" application framework. Although it seems silly to dynamically load hardware interfaces (after all, we do not expect the hardware to change during execution), this is necessary if we wish to distribute both binary and open source versions of our application frameworks. The requirement for dynamic loading of binary components is a key factor that influences the design choices in Section III-B.

Finally in section IV we provide an illustrative example of writing a servo control algorithm that requires velocity feedback that may or may not be available from the hardware. Our approach is contrasted with other design options that are based on class inheritance.

A. Unified Task and Device Interface

Figure 1 shows a typical simple hierarchy of "loops" required in a robot controller. It is quite common for these "loops" to run either on COTS hardware or be implemented in software. Consider the example of two different I/O boards: one that provides on board servo control by embedded hardware and the other which is a simple I/O device, with servo control implemented on the main computer. For concreteness, let us assume that in the former case, the supervisory control loop must call a function that sets some I/O bits; in the latter case, the supervisor must push the desired position onto a queue (FIFO/IPC), which is later de-queued by the servo control thread or process. Our requirement is that the two cases be transparent to the supervisory control loop. We have two types of abstractions, based on the whether the implementation uses a software periodic task, as in the case of SoftwareServo, or wraps a hardware device, as in the case of HardwareServo. We call these abstractions Task and Device, respectively. In order to facilitate interchangeability, we have adopted a class hierarcy where the Task class inherits from the Device class.

Moreover, if the request on the task or device is encapsulated as an object, then certain actions can be centralized in that encapsulation object. An example, which we will elaborate in next few paragraphs, is the case of queueing the requests to the task. We used the command pattern [12] to provide us with this ability and to allow us to specify, queue, and execute requests at different times.

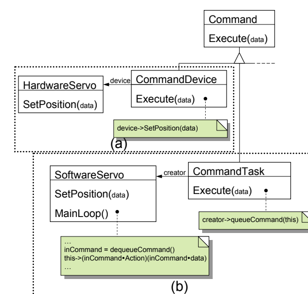

Fig. 2. A hierarchy of command objects used on (a) a device (b) a task

Figure 2 shows the class hierarchy of the Command class which is used to implement the command pattern. The Command class is an abstract class that is specialized further depending on whether the command needs to behave as a message that is queued. CommandDevice class objects are used for Devices and CommandTask class objects are used for Tasks. This implementation is different from Functors, objects that behave as functions, in that the Command object also maintains a binding between the receiver and corresponding action. This allows a degree of transparency.

The basic flow of control is shown in Fig. 3 for the case where a higher level task (trajectory loop) interfaces with a lower level device (external servo controller, *HardwareServo*). Because tasks and devices share a common base class, we could just as easily have used a lower level task (*SoftwareServo*) instead of a device. Sometime during initialization, both the higher level task and the lower level device are

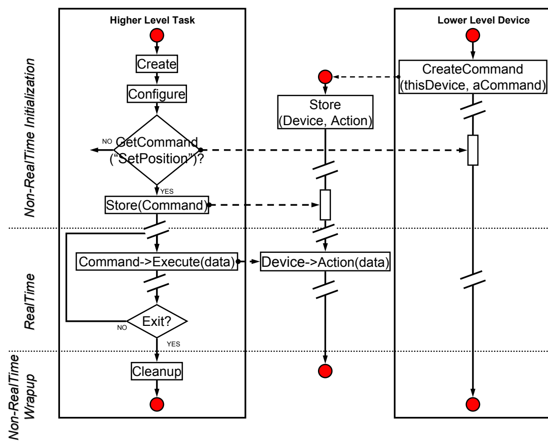

Fig. 3. Timeline of main objects when lower level is a device

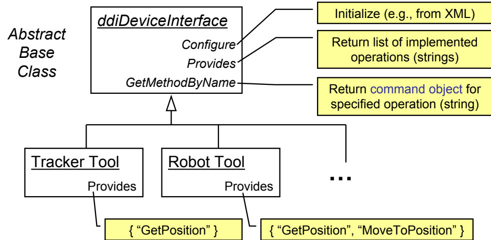

Fig. 4. A hierarchy of device interface objects

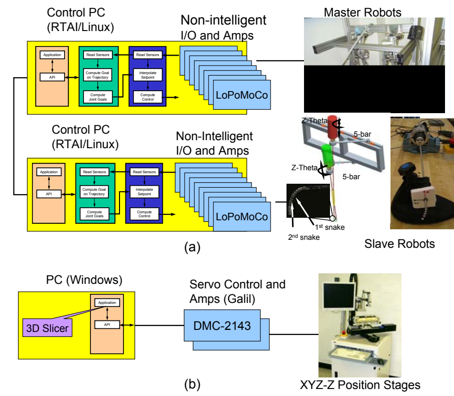

Fig. 5. Examples of alternatives for a CIS system (a) A complex teleoperative system with large number of Dof's (b) A simple 4 axis positioning robot

created. These in turn create a number of command objects, each corresponding to the functionality they provide. For this example, both the SoftwareServo and HardwareServo would create command objects corresponding to SetPosition, though the implementation of each of these methods would differ. At a later time instance, still in the non-realtime phase, the higher level task would query the lower level device, and if available, obtain a command object and store it for later use. We shall defer the discussion of the exact query mechanism to section III-B.

In the real-time phase, the higher level task can call the Execute method of the stored command object to carry out the necessary operation. The higher level task does not know which subclass of the command object it is using. For example, the SetPosition command of the SoftwareServo task places the request on the queue of the MainLoop. More importantly, it also places the data to be operated upon on the queue in an address space independent way. The SetPosition command of the HardwareServo device is different – it simply calls the actual method of the specific instance that created that particular command.

B. Organization of Devices

Sometimes a single hardware or software module does not fulfill all the necessary requirements of the higher level modules. The situation becomes complex when one wants to provide families of alternative, parameterized objects to compose different systems. Consider the case of providing a façade to a complex robotics system that requires multiple I/O boards because the robot has more actuated joints than the number of axes on a single board. Moreover, the user may wish to add hardware for other sensors or devices. Figure 5(a) shows the case where more than one I/O board must be associated with the servo controller [2]. Moreover, the "trajectory" loop in this case is based on force control, which requires an additional force/torque sensor. Figure 5(b) shows a simple robotic positioning system that was developed using primarily COTS components[1]. The question is how to handle this explosion in number of combinations, with minimal change in code, while at the same time allowing easy addition of new features.

The two solutions to the above problem are:

1) Use of C++ support for static configuration such as multiple inheritance or aggregated objects.
2) Use of a "dictionary" of command names and command pattern to augment dynamic C++ features such as Run-Time Type Information.

We discuss both approaches in this section and explain why we chose the second approach.

In the first approach, the user can create new objects by extending existing classes or by creating new classes using aggregation or composition. The major liability of composition is that it complicates object hierarchies and leads to code "blobs" or coexistence of disparate qualities, identities, or methods in a new class. On the other hand, in case of aggregation the user has to write many forwarding wrappers for the new class. This is not only tedious, but also impairs adaptability and maintainability.

A solution to this problem is to organize object-oriented frameworks as a family of configurable layers or *Mixins*. The details of these are beyond the scope of this paper, but the reader is refered to [4]. Jung et al [13] made use of class templates containing member classes to implement these layers or Mixins. Though templated classes can become cumbersome to write and debug as the number of parameters increases, the authors mention using tools developed in [14] to ease the programming.

As mentioned earlier, we require a dynamic system. Furthermore, there is the question whether all possible components are designed before runtime, and if the dynamic reconfiguration only takes place within the statically defined combinations. Again, the requirement of dynamically loadable binary components dicates that our system provide maximum flexibility without increasing the number of components to

distribute. The template based approach allows for dynamic reconfiguration but is restricted to the combinations designed at designtime.

We focus on the second approach to implement the same concept of building the combinations from aggregates (see Fig. 4). Our approach continues to take advantage of the command pattern to allow a dynamic, yet efficient, method for generating aggregates. First, we associate a string name with each command object created by the device. Command objects associated with similar methods implemented by different tasks or devices are given the same string, thus creating a "dictionary" of names. For example, Fig. 4 illustrates that a robot tool and a tracker tool both implement a "GetPosition" command. We also developed a mechanism for dynamic interface query, where a caller can query an object for the methods (string names) it Provides.

To facilitate use of a combination of different devices, we provide a mechanism to *concatenate* devices into composite devices. The composite devices are divided into two categories: *homogeneous* and *heterogenous*. Homogeneous components are formed when one or more types of identical devices are concatenated together; an example is when multiple I/O boards are needed to control a large number of axes. An example of a heterogenous device is one that is formed by combining a force sensor device with a motion controller, for example to create a robot with simple admittance type control based on force applied by the user. This difference is transparent to the higher level layers that use the heterogenous or homogeneous device.

For the heterogenous devices, the list of commands provided is a simple concatenation of the list of commands provided by its constituents. The homogeneous case is handled by the use of a composite command which is shown in Figure 2. This allows us to evolve and adapt in ways that are not (completely) determined at design time. Another advantage of using composite devices is that it allows us to use a system architecture similar to the levels shown in Figure 1. Thus, we can reduce the model to a stacking model. In other words, in our architecture each higher level has one and only one lower level. This simplfies the query mechanism as the higher level device has to query only one device.

Though both approaches allow any device to be used in any context, provided that the device meets the conditions and restrictions required for that context, the use of class templates as a means of creating new devices ensures that any incompatibility in the condition or requirement is caught at compile time. Unfortunately, the dynamic nature of this design implies that some errors, such as attempting to use a device interface that does not implement a required command, can only be detected at runtime. Though this approach is more flexible than using a class hierarchy, it requires careful maintenance of, and adherence to, a "dictionary" of command names and their associated data types.

Runtime Adaptation of Devices: An added advantage of our approach is that it gives us the ability to dynamically discover methods and attributes of classes at runtime and add methods

Listing 1. OO approach by creating a common interface class

```
1  class IOBase {virtual double Velocity() = 0;};
   class BasicIO : public IOBase {
   protected: //define state
   public:
   double Velocity() { //compute using difference}
6  };
   class IntelligentIO: public IOBase {
   public:
   double Velocity() { //do I/O to hardware }
   };
```

using dynamically loadable modules. At the same time, the use of the command pattern ensures that the the code is efficient at least for time-critical sections. We illustrate this by an example in section IV.

IV. Test Example

In this section we consider the following illustrative example of writing a simple servo controller using two different I/O boards, one that provides direct measurement of velocity and another that does not. In the absence of a direct velocity measurement, we locally compute an estimate from the measured positions. There are a number of ways in which one can write a generic servo controller that is independent of the underlying hardware. Before presenting our approach, we present some existing software principles and their approaches to this problem and discuss their merits and demerits.

OO frameworks typically provide base classes which represent the infrastructure common to a number of applications. The purpose of the interface layer is to provide a common interface to the different implementations. The advantage is that this isolates the program from differences between implementations. The pseudo code for this approach is given in Listing 1.

There are two main drawbacks with this approach, the first being that such code leads to fat interfaces or code bloat interfaces which represent the union of possible features provided by all robots. From our experience with existing software, the interface classes end up as a subset of features that are available on all devices, plus a few methods that can be easily emulated in software. This problem is more common as the devices get more complex with widely varying interfaces, such as optical tracking devices. The second drawback is that the IO class needs to maintain a state to be able to emulate the missing methods, which in some cases might be redundant as the user of this class might also store the same information. Moreover, in the case of complex devices, not all features might be used by every user of the class; this would lead to needless storage and overhead.

Another OO design pattern deployed to solve such problems is to combine and extend some base classes in various ways with the required specialized algorithms and use C++ features such as RunTime Type Information (RTTI) to discover the type of an object at runtime. The pseudo code for the velocity example for this approach is given in Listing 2.

Due to limitations of C++, one has to check for all possible class types that can provide the specific method before making a decision, which can be tedious and error prone. The more

```
class IOBase {};
class BasicIO : public IOBase {};
class IntelligentIO: public BasicIO {
public:
    double Velocity() {//do I/O to hardware}
};
class MotionServo {
public:
    IOBase *device;
    MotionServo() {
        // dynamically create the underlying
           hardware device
        device = Create(deviceType);}
    double computeOutputVoltage() {
    if (dynamic_cast < IntelligentIO *>device) {
        // device has specific features
        (dynamic_cast < IntelligentIO *>device)->
           Velocity()
    } else {//compute using difference}
  }};
```

Listing 2. Using RunTime Type Information (RTTI)

```
class MotionServo {
    Device *device;
    Command *velocity, approximateVelocity;
    void NonRealTime() {
        velocity = device->GetMethodByName(''
           velocity'');
        if (velocity == Command::NOPS) velocity =
            approximateVelocity;
    }
    void RealTime() {(*velocity)();//do servo
      control}
};
```

Listing 3. Using command pattern to achieve flexibility

general case of this approach would involve multiple inheritance, where the parent classes define the set of interfaces. This approach is well suited if one is not concerned by efficiency of the runtime code.

More recently, Jung et al [13] use partial template specialization in the C++ compilers to generate code that has better efficiency and smaller executables. The basic idea is a static IF statement, which uses partial specialization to return a "then" class if a certain condition is true and an "else" class if the condition is false. Unfortunately, templates are statically type checked at compile time. This does not allow for runtime modifications to be made, making the idea of dynamically loadable modules difficult to implement.

Listing 3 presents the same example using our approach. In our approach the code has been split into two segments: the non-time critical and time critical. In the non-time critical segment, we use the mechanism discussed in section III-B to check if the Velocity command object is available in the underlying device. If not, a local command object is stored. In either case, the time critical segment uses a meaningful command object and sees no difference.

V. CONCLUSION

Even though CIS systems exist in the research and commercial communities, progress is hampered by the lack of open source software that can be certified for clinical use. We endeavor to address this need by making the *cisst package* available as open source software in the near future. Presently, some components of the *cisst package* are available as open source software at *http://www.cisst.org/cisst/*, whereas other components and the frameworks are in active development.

In this paper we enumerated some of the robot controller requirements that are particular to our research in this area: they are created from a number of devices with different features, include components other than robots, and must be certified for clinical use. The *cisst package* will provide *source code components* that are assembled into a family of *application frameworks*. Researchers can dynamically load hardware-specific and application-specific *binary components* into the framework to satisfy their particular system and application requirements. Some key design features are the interchangeability of tasks and devices (provided by the inheritance structure), the rigorous application of the command pattern and the creation of a dynamic interface query mechanism based on command names (strings). Future work includes middleware to support command transfers between tasks residing on different machines, which should be facilitated by the encapsulation provided by the command pattern.

ACKNOWLEDGEMENT

We thank Prof. Russell H. Taylor for his valuable comments and support of the cisst library. We also thank Ofri Sadowsky, who is the primary author of the cisstVector package used extensively within our robot control framework. Development of the cisst software is supported by NSF grant EEC9731748.

REFERENCES

[1] J. Li, E. Balogh, I. Iordachita, G. Fichtinger, and P. Kazanzides, "Image-guided robot system for small animal research," in *Intl. Conf. on Complex Medical Eng. (CME 2005)*, Takamatsu, Japan, 2005.
[2] A. Kapoor, N. Simaan, and P. Kazanzides, "A system for speed and torque control of dc motors with application to small snake robots," in *Mechatronics and Robotics*, Aachen, Germany, 2004.
[3] R. Taylor, P. Jensen, L. Whitcomb, A. Barnes, R. Kumar, D. Stoianovici, P. Gupta, Z. Wang, E. deJuan, and L. Kavoussi, "Steady-hand robotic system for microsurgical augmentation," *Intl. J. Robotics Res.*, vol. 18, no. 12, pp. 1201–1210, 1999.
[4] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. Addison-Wesley, 2000.
[5] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
[6] R. E. Johnson, "Frameworks = (components + patterns)," *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, October 1997.
[7] E. Ehlinger, "Creating truly maintainable class factories," *C/C++ Users Journal*, vol. 18, no. 11, pp. 22–33, November 2000.
[8] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the orocos project," in *Proc. IEEE Intl. Conf. Robot. and Automat.*, Taipei, Taiwan, 2003, pp. 2776–2771.
[9] A. Brooks, T. Kaupp, A. Makarenko, and S. Williams, "Towards component-based robotics," in *IEEE/RSJ Intl. Conf. on Intell. Robot. and Sys.*, Edmonton, Canada, 2005.
[10] K.-U. Scholl, V. Kepplin, J. Albiez, and R. Dillmann, "Developing robot prototypes with an expandable modular controller architecture," in *Proc. of the Intl. Conf. on Intelligent Autonomous Systems*, Venice, 2000, pp. 67–74.
[11] H. Peters, J. Raczkowsky, and H. Woern, "Approach to an architecture for a generic computer integrated surgery system," in *IEEE/RSJ Intl. Conf. on Intell. Robot. and Sys.*, Edmonton, Canada, 2005.
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.
[13] E. Jung, C. Kapoor, and D. Batory, "Automatic code generation for actuator interfacing from a declarative specification," in *IEEE/RSJ Intl. Conf. on Intell. Robot. and Sys.*, Edmonton, Canada, 2005.
[14] D. Batory, "Feature-oriented programming and the ahead tool suite," in *Intl. Conf. on Soft. Eng.*, Edinburgh, Scotland, 2004.