International Conference on Robotics and Systems (IROW) 2005, Edmonton, Canada

Automatic Code Generation for Actuator Interfacing from a Declarative Specification

Ed Jung and Chetan Kapoor[1], Don Batory[2]

Robotics Research Group, Dept. of Mechanical Engineering[1]
Product Line Architecture Research Group, Dept. of Computer Science[2]
University of Texas at Austin
Austin, TX

ed.jung@mail.utexas.edu, chetan@mail.utexas.edu

***Abstract* - Common software design practices use object-oriented (OO) frameworks that structure software in terms of objects, classes, and package; designers then create programs by inheritance and composition of classes and objects. Operational Software Components for Advanced Robotics (OSCAR) is one such framework for robot control software with abstractions for generalized kinematics, dynamics, performance criteria, decision making, and hardware interfacing. Even with OSCAR, writing new programs still requires a significant amount of manual labor. *Feature-Oriented Programming* (FOP) is method for software design that models and specifies programs in terms of features, where a feature encapsulates the common design decisions that occur in a domain. A set of features then forms a domain model for a *Product Line Architecture*. Product variants in this product line can then be generated from a declarative specification. FOP and related technologies are emerging software engineering techniques for automatically generating prorams. Our research applies FOP to robot controller software. As an example, the domain of hardware interfacing is analyzed and 41 features identified. A GUI for specifying and generating programs is presented as well. Analysis of features shows 200 possible different programs could be generated.**

Index Terms – Robotics. Product Line. Feature Oriented Programming, Generative Programming.

I. Introduction

The *Robotics Research Group* (RRG) at UT-Austin has developed the *Operational Software Components for Advanced Robotics* (OSCAR) framework [16][20]. The framework addresses the integration of generalized kinematics, dynamics, performance criteria, decision making, hardware interfacing, and manual controllers into robot controller software. Experience with OSCAR has shown that OO methods improve productivity, reusability, and comprehensibility of robot software. Applications, however, must still be manually written, with each new application being a one-off effort, prone to errors.

Among the reasons for these continuing problems are the low level of granularity at which components are designed, and the lack of explicit guidance for assembling those components. In other words, rapid software assembly requires not just components, but a systematic method for assembling those components.

Feature Oriented Programming (FOP) is one such method for rapid assembly of software [5]. The success of FOP relies upon the assumption that *application software in a domain is so well understood, that software construction can be automated*. RRG has written robot control software for several different robot applications [22][19]. Many of these tend to share some common set of base code, as well as design decisions. These programs thus form different *product variants* in a *product line*.

FOP differs from OOP in that *features*, rather than classes, objects, or packages, form the fundamental units of software. A feature is a domain-level abstraction, or some capability for the software that is significant to the end user. A feature encapsulates any code (including documentation, test programs, etc.) necessary to implement that capability. Programs may then be specified by their desired features, in a declarative specification.

RRG is applying FOP methods to robot control software, in collaboration with the Product Line Architecture Research Group [21] in the Department of Computer Science. The goals of this paper are to introduce some basic techniques for implementing FOP, and their application to robot software. A GUI for building specifications is presented.

The goal of this work is to introduce FOP, contrast FOP methods to OOP methods, and demonstrate a sample application of FOP to robot software. The novelty of this work lies in its application of FOP to automatically generate code from a declarative specification.

II. Robot Software Frameworks

A. *Object Oriented Methods and Frameworks*

Object oriented frameworks and design patterns are the dominant methods for software design today. Among recent efforts are frameworks such as QMotor and QRobot, oriented towards servo control, and robot control respectively [16][8]. The Orocos projects, Open Robot Control Software and Open Realtime Control Services, are open source efforts [6] which rely upon design patterns [10] as well as OO methods. The OSCAR framework has been under continuous use and development at RRG for several years [16][20][16]. Users have included NASA, and DOE [19].

OO frameworks typically provide a set of base classes, which represent the infrastructure common to a number of applications. These base classes may then be extended by inheritance and polymorphism, or combined through aggregation or composition, to implement specific applications. *Design patterns* recognize that some design problems occur commonly, in a common context, and have a common solution [12]. They may be used as a starting point for the design of a new application. Three examples of OO design will be considered---from OSCAR, QMotor/QRobot and the Measurement Systems Framework case study [5]. Each covers a different scope of software (robotics algorithms, actuator interfacing, and sensor interfacing, respectively). The intent here is to demonstrate the application of OO techniques to software for robotics and automation.

B. OSCAR

Though OSCAR is specific to RRG, the fundamental ideas are common to most OO frameworks. That is, the use of inheritance, aggregation, and composition within a common structure, to create specific applications. QMotor/QRobot provide additional examples of OO methods. The Measurement Systems Framework employs OO methods as well as design patterns.

OSCAR consists of a set of class libraries, some of which are described in Table 1. These classes may be combined or extended in various ways for specific applications. For example, IKJacobian and IKPuma classes extend the base kinematics classes with specialized algorithms.

Though these classes are flexible and reusable, they must still be manually combined to create a new application. In practice, new applications are often created by copy/paste, or based on the past experience and bias of a developer. Clear application patterns, however, exist; these are further enforced by the uniform interface to OSCAR classes and the data flow inherent to robot control applications. Ideally, the construction of these patterns and applications should be automated to reduce the error and variability introduced by manual construction.

C. QMotor/QRobot

The *QMotor RTK (Robotic Toolkit)* [17] is a software package with similar goals to the OSCAR Device Domain. QMotor is capable of interfacing with the *Puma 560, Barrett WAM (Whole Arm Manipulator),* and *IMI Direct Drive* robot. The approach for adapting robots into QMotor uses traditional OO tehcnqiues. A base class contains common code; different extensions implement robot specific code. An example of extension through inheritance is presented in Table 2 for a Puma.

Table 1. OSCAR Domains

Domains	Description
Decision Making	Algorithms for criteria based decision making
Device	Standardized interfaces to robots, sensors, actuators
Dynamics	LaGrange and Newton-Euler dynamics algorithms
Forward Kinematics (FK)	Position, velocity and acceleration level
Inverse Kinematics (IK)	Numerical, closed form and redundancy resolution
Motion Planning	Joint space and end effector motion planning and curve criteria
Obstacle Avoidance	Criteria based obstacle avoidance algorithms
Performance Criteria	30+ different criteria for manipulator control

Note that although the code for the PID algorithm might be reusable with other robots, it is embedded within the `PumaPIDControl` class, and cannot be reused.

This common PID code could be refactored into the base class, `ManipulatorControl`. Any class deriving from `ManipulatorControl` would then be able to access the PID code. Such approaches tend to lead to “fat” interfaces, which represent the union of possible features or capabilities provided by all robots.

Table 2. QMotor Pseudo-code

```
Class PumaControl : public ManipulatorControl
{
  // Puma specific implementation added here
  // inherits calculatePositionControl()
  // from ManipulatorControl
};

class PumaPIDControl : public PumaControl  {
public:
  void calculatePositionControl()  {
   //use PD calculation
   //  from Manipulator Control

PumaControl()::calculatePositionControl();
  //foreach joint
  //  calculate integral term, add to PD term
  }
};
```

C. Measurement Systems Framework

The case study conducted by [5] employs two common design patterns in the design of a measurement systems framework—the *Strategy* pattern and the *Factory* pattern [12]. The framework controls a relatively simple manufacturing system. Sensors measure some property of an incoming object (e.g., weight, size) on a conveyor; if the value is acceptable, the object passes. If not, an actuator rejects the object. Only the *Strategy* pattern will be considered here.

The *Strategy* pattern decouples a class from a specific implementation by factoring the implementation into an independent *Strategy* class. The original class may then delegate calls to an instance of the *Strategy*.

In the measurement framework, sensors may use one of several *CalibrationStrategies*, *CalculationStrategies*, and *UpdateStrategie*s. The *UpdateStrategies*, for example, determined how and when a sensor was updated. These included a *Client* update strategy, a *Periodic* update strategy, and an *OnChange* update strategy, which updated the sensor value upon a client invocation, at a periodic interval, or any time the sensor value changed, respectively.

The authors of the case study [5]noted as a major benefit of the *Strategy* pattern a “dramatic increase in flexibility.” New implementations could be added simply by writing a new Strategy class. One major liability was that it “dramatically complicate[d]” object interactions, as *Strategy* objects and parent objects had to be manually bound each other.

This liability is evidence of “object schizophrenia” [9], meaning that when objects are broken into smaller fragments to implement a pattern, they must be recomposed into a single object. These fragments suffer from “schizophrenia” because they have no reference to the identity of the whole. The author of the case study in [5] estimates that up to two thirds of application code involved the binding of objects to objects.

The problems stated above are a consequence of program fragmentation, as design patterns tend to fragment an application into many “little classes” and “little methods” [9] . One negative consequence is that the level of granularity becomes much lower, making more difficult both application construction and maintenance.

D. Summary

The application of OO methods to robot software is commonplace. Robotics frameworks also tend to identify the same kinds of high level abstractions. In other words, *any* robot software framework besides OSCAR would most likely identify kinematics algorithms, motion planning algorithms, dynamics algorithms, etc. as abstractions, and implement them as classes. Similarly, any framework for sensor or actuator interfacing, besides QMotor or the measurement framework would likely identify different PID algorithms or hardware (e.g., sensor, actuator, tool) as abstractions, and implement them as classes.

Hooking these abstractions together into a specific application is the most difficult part of building robot software. This is typically accomplished by (1) extending existing classes, (2) creating new classes, and (3) using aggregation/composition to connect objects. During these steps, errors may be introduced into an application. What is desired is a way to systematically automate the process such that they are reliably repeatable.

The focus here is not on the capabilities of OSCAR, but the application of FOP to automatically assemble OSCAR code. The remaining sections introduce FOP, and apply it to the sample domain of actuator interfacing software.

II. Feature Oriented programming (FOP)

FOP is a technique aimed at the design of *product lines* for software. A product line is a set of applications which are variants of a single or a few common applications. Thus the members of a product line can be built from a *common set of components*, using a *systematic* method of assembly. In contrast, frameworks tend to employ an informal, ad-hoc approach to assembling applications. FOP further makes possible the *automatic generation* of applications.

One of the fundamental assumptions of FOP is that *the applications in a specific domain are understood so well, that the steps to build them can be automated*. This is exactly the case for OSCAR applications at RRG.

Features are the fundamental unit of modularity in FOP, rather than classes, objects, or packages. Features represent high level, domain specific abstractions, relevant to a domain expert. Features also encapsulate software fragments needed to implement the feature. A specific application program may then be specified in terms of its features. These features, which represent fragments of code, are then composed together to build a specific application. At the most advanced level, a program can be specified from a GUI, using checkboxes and lists, similar to how a computer may be ordered from the Dell website.

One way to consider features is as a series of *step-wise extensions*, where each features builds upon an existing program by adding some functionality. A complete, existing program may be augmented by a feature, giving it additional capabilities.

The full theory and technique for implementing FOP are beyond the scope of this paper, but the fundamental idea is *feature refinement*. Detailed explanation of FOP and related program composition technologies is available elsewhere [21][9]; the immediate purpose here is to demonstrate some basic techniques and advantages of using FOP. *Mixins* or *mixin layers* are one straightforward implementation of refinements [23].

B. Implementing Refinements with Mixin Layers

Mixins are a technique for implementing feature refinement that requires only a C++ compliant compiler with template support. A *mixin* is a refinement of a single class, a *mixin layer* is simultaneous refinement of several classes. Table 5 is

Table 5. Puma Mixin Layers Pseudo-code

```
class PumaServos  {virtual void setPosition()
   {/*Puma specific set position*/}  };

template <class parent>
class RobotInterface : public parent  {
public:
 class ControlAction         {
 public:
    void calculatePositionControl()=0;
    RobotInterface * pRobotInterface;
 };

 virtual void setPosition()  {
     m_pCaObject->calculatePositionControl();
     parent::setPosition();
 }
 ControlAction *m_pCaObject;
};

 // PID control feature
template <class parent>
class PID : public parent  {
public:
  class ControlAction
       : public parent::ControlAction  {
   void calculatePositionControl()
        {/*do pid law*/}
   float Kp, Ki, Kd;
 };
};

 // Fuzzy Logic Control feature
template <class parent>
class FuzzyLogic : public parent  {
public:
 class ControlAction
       : public parent::ControlAction  {
   void calculatePositionControl()
      {/*do fuzzy law*/}
   float high, medium, low;
 };
}

typedef PID<PumaServos> PIDPuma;
typedef FuzzyLogic<PumaInterface> FuzzyPuma;
```

an example of *mixin layers* on the Puma code.

Note that a feature refinement may extend classes (lines 4, 21, 23, 33, 35), add classes (line 6), or (3) aggregate objects (lines 16, 27). Feature refinement is thus a technique for automating framework extension.

Mixin layers using templates can be difficult to debug for very large systems. Therefore, special tools have been developed by [21] as part of the *AHEAD (Algebraic Hierarchical Equations for Application Design)* tool suite [4]. Details of the AHEAD tools, and AHEAD theory, which underlie this paper, are available at [21].

III. Features In Robot Software

Domain modelling is the process of identifying features in software for a specific domain. Thus it is necessary to identify the different features of robot applications.

Application software in OSCAR is typically divided among three layers—an upper layer for communicating with Human Machine Interface (HMI) devices, a middle Computational Components (CC) layer, and a lower Device Interface (DI) layer.

The upper HMI layer interfaces with joysticks, manual controllers, Spaceball, GUIs, or any other software used to interface with a human operator.

The middle CC layer consists of computational algorithms and decision making software for robot control, such as kinematics, dynamics, performance criteria, motion planning, etc., and supervisory control of the DI layer.

The lower DI layer is used to communicate with different sensors and robot hardware, as well as to isolate the CC layer from differences among hardware.

The division of software among these layers is based roughly on timing requirements, where the DI layer requires real time control, the timing of the CC and HMI layers is less stringent. The focus of this paper is on the DI layer.

A. Device Interface Features

The first purpose of the DI layer is to provide a common interface to the different kinds of robots at RRG. This interface should isolate an application program from the differences between robots; the DI layer may be considered as a kind of Hardware Abstraction Layer (HAL) for robots. Features related to this purpose are termed Actuator Hardware features.

The second purpose of the DI layer it to provide real time capabilities. Different types of robot actuators may have different timing requirements, i.e., each robot may require joint commands to be sent at a different rate. The DI layer should provide commands to the robot's embedded controller at a consistent rate, and serve as a buffer between the CC layer and the robot controller in case the CC layer cannot meet the required rate.

B. Actuator Hardware Features

The Actuator Hardware features represent the different capabilities of the robots used at RRG. These robots are the KB2017 dual arm robot, the Powercube modular robot, and the Roboworks simulation environment, which is a virtual robot interface. Table 6 summarizes these robots.

Each robot directly supports various features in hardware, such as *joint command modes* (i.e. Position, Velocity, Current, Torque), *joint range limits* (i.e. joint position limits, joint velocity limits, etc.), *joint excess limits* (i.e., position excess limits, velocity excess limits, etc.), and different communications protocols.

The *range limits* and the *excess limits* differ in that the range limits features trap joint commands outside an acceptable range, while the excess limits features trap *changes* in joint commands that exceed an acceptable value.

Emulation of features is also desirable, if they are not directly supported by hardware. The most obvious example is different types of joint limits, which can be easily emulated in software. Redundant joint limits (both hardware and emulated) are also possible.

Some control types may also be emulated, such as position or velocity control. For example, the Roboworks environment only accepts position commands. It may be used to simulate the KB2017, which supports velocity commands. In such a case, a velocity control mode might be emulated in Roboworks, by calculating a derivative. There are further real time features necessary to emulate velocity, which are discussed in the next section.

Some amount of manual coding will always be necessary to create an interface to a robot. For the examples above, the

Table 6. Features Supported by Robot Hardware

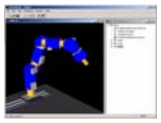
Robot	Dof	Features
KB2017	17	-Position, Velocity, Torque Control -Position Range Limits -Position Excess Limits
Powercube	1-7	-Position, Velocity, Current Control -Position, Velocity, Current Range Limits -Position Excess Limits
Roboworks 	n	-Position Control

Table 7. DI Feature Summary

PCoInt VCoInt CCoInt TCoInt	PRaInt VRaInt CRaInt TRaInt	PExInt VExInt CExInt TExInt
PCoEm[*] VCoEm	PRaEm VRaEm CRaEm TRaEm	PExEm VExEm CExEm TExEm
PCoHw VCoHw CCoHw TCoHw	PRaHw VRaHw CRaHw TRaHw	PExHw VExHw CExHw TExHw
P = Position, V = Velocity, C = Current, T = Torque *Co = Control, Ra = Range Limits, Ex = Excess limits* *Int = Interface, Hw = Hardware, Em = Emulated*		

[*] Torque and current control execution are theoretically possible, but would require full inverse dynamic models, which would negatively affect the timing of the DI layer.

KB2017, Powercube, and Roboworks all use different communications protocols and different Applications Programming Interfaces (APIs). The goal for the DI layer is to minimize this amount by reusing common features.

Considering the above three robots, the set of common features for Actuator Hardware is summarized in Table 7, where the control types are abbreviated.

Also, each of the above features may be supported by hardware or emulated in software. Both the emulated and the hardware version of a feature may be desirable, as with *position range limits*. In other words, there is a single common interface to the above features, with different implementations. Thus *there are three different versions* of each feature in Table 7. For example, there is a *Position Control Interface* feature, a *Position Control Hardware* feature, and *a Position Control Emulation* feature.

C. Real Time Features

The real time features were mentioned earlier in relation to emulated velocity control. To emulate velocity control, the DI layer must run at fixed rate to accurately calculate the derivative of position. Several *RealTime features* are needed to accomplish this task, which adds code for multi-threading and synchronization. These features are summarized in Table 8. More detailed explanation is available in [13][18].

Table 8. Real Time Features

Feature	Description
`Locking | Nonlocking`	Thread-safe or not thread safe code
`Active | Passive`	Multi or single-threaded code
`AvgTuning | DefaultTuning`	Different policies for adjusting execution rate
`MotionTime`	Time step for actuator commands
"A | B" *means choose one of A or B*	

Briefly, the `Locking | Nonlocking` features allow a user to choose between thread-safe and non thread-safe versions of the DI layer, the `Active | Passive` features allow the user to choose between a standardized multi-threading mode, which allows DI to execute asynchronously, or a single-threaded mode, in which case DI executes synchronously with the CC layer. The Tuning | Nontuning features provide different policies for changing the execution rate of a multi-threaded DI program, and the MotionTime feature simply adds a data member for storing the time step between actuator commands.

To emulate a velocity control mode in Roboworks, the set of *Real Time features* {`Locking, Active, MotionTime, FixedTuning`} is required. Alternatively, a user might specify the feature set {`Locking, Active, MotionTime, AvgTuning`}. If the standard multi-threading mode provided by `Active` is not appropriate for a given application, a custom version of the Active feature can be implemented. Alternatively, users could simply specify the *Real Time feature* set {`Locking`}, which would make DI thread-safe, and implement their own concurrency code on top of the DI program.

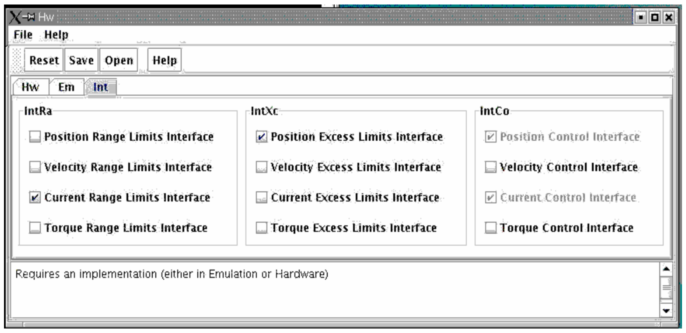
IV. Declarative Specification

Given many features in a domain, there are potentially thousands of different combinations, each of which specifies a different program. This explosion in the number of possible combinations is known as *feature combinatorics*. If there are n optional features, there are 2^n possible different combinations of those features.

Some of these combinations will be invalid or undesirable. Furthermore, some features may impose constraints or requirements on other features (e.g., Position Range Limits Interface (`PRaInt`) requires the Position Command Interface (`PCoInt`)). It is necessary to automate the application of such rules in an easy to use interface. In AHEAD, this is accomplished with a grammar and design rules.

The role of a grammar is to specify sequences of features to be composed, and design rules guarantee that the features in a sequence are compatible. From such a grammar a declarative domain-specific language can be created, with a domain-specific editor that allows robot controller programs to be specified declaratively.

The *guidsl* tool in the AHEAD tool suite can automatically generate GUIs from a grammar; the GUI of Figure 1 was generated from the DI grammar. The *guidsl* tool simply takes a text file containing the grammar as an input. This text file



IntRa
- [ ] Position Range Limits Interface
- [ ] Velocity Range Limits Interface
- [✓] Current Range Limits Interface
- [ ] Torque Range Limits Interface

IntXc
- [✓] Position Excess Limits Interface
- [ ] Velocity Excess Limits Interface
- [ ] Current Excess Limits Interface
- [ ] Torque Excess Limits Interface

IntCo
- [✓] Position Control Interface
- [ ] Velocity Control Interface
- [✓] Current Control Interface
- [ ] Torque Control Interface

Requires an implementation (either in Emulation or Hardware)

Figure 1. Declarative GUI for Device Integration

additionally contains simple design rules and annotations to provide user help. As an output, it produces a GUI with a sequence of tabs and checkboxes. The text box at the bottom of Figure 1 provides help and information about a specific option when a user hovers the mouse pointer over a feature.

The *guidsl* tool uses Propositional Satisfiability (SAT) algorithms to generate the logic behind the GUI. Such algorithms are used for design automation and constraint satisfaction problems in other domains. Work on the *guidsl* tool and its use is currently in progress, and interested readers should refer to the AHEAD documentation and website [20].

Because the DI domain is relatively simple, it is possible to exhaustively enumerate the total number of valid configurations. There are 6 possible configurations of the *Real Time* features, due to the constraints between features. Two of these configurations are for single threaded versions, and 4 are for multi-threaded versions. The number of possible *Actuator Hardware* configurations varies with the specific robot. Table 9 summarizes the 200 possible valid configurations.

IV. Related Work

Table 9. Features Supported by Robot Hardware

Robot	Actuator Hardware	Real Time	Total = Actuator Hardware * Real Time
Roboworks	4	4	16
Roboworks RT	20	2	40
KB2017	12	6	72
Powercube	12	6	72
		Total	200

FOP shares several concepts in common with other work in software engineering and generative programming. FOP is a method for designing *product line architectures*; PLA methods focus on designing software components for building sets of product variants which cover a domain.

FODA (Feature Oriented Domain Analysis) was among the earliest methods for designing software in terms of features in a domain.

Aspect Oriented Programming (AOP) is a design method which is currently gaining momentum [10]. AOP augments traditional OO methods with *aspects*, where an aspect is a code fragment which can be non-invasively inserted, or "weaved," into an existing code base.

Agile methods are another recent movement gaining popularity, and have the notion of *feature driven development [11]*. This commonality with FOP is coincidental, however, and it could be argued that agile methods and PLA methods (including FOP) are completely opposed. PLAs assume that software in a domain is so predictable and well understood, that a systematic set of rules for design and assembly of components can be defined. Agile methods assume that software design is so unpredictable that *no such systematic design* is possible.

V. Conclusion

Feature Oriented Programming (FOP) is a technique for designing software product lines. In contrast to object oriented techniques, FOP models programs in terms of features, which operate at a much higher level of abstraction. Programs may be specified from a set of features, and then automatically generated.

Initial results using FOP for the automatic generation of robot hardware interfacing software were presented. A feature model, consisting of 41 *Actuator Hardware* features and *Real Time* features, was used to model this software. Examples for specifying two different program variants were also presented. Exhaustive enumeration of configurations results in a total of 200 possible programs for interfacing with robot hardware.

References

[1] Batory, D.; Geraci, B.J., "Composition validation and subjectivity in GenVoca generators," IEEE Transactions on Software Engineering, Vol. 23, No. 2, Feb 1997, pp 67-82.

[2] Batory, D., Cardone, R., and Smaragdakis, R. "Object-oriented frameworks and product-lines," 1st Software Product-Line Conference, Aug. 1999, Denver, Colorado.

[3] Batory, D., Sarvela, J.N., and Rauschmayer, A., "Scaling step-wise refinement," International Conference on Software Engineering, May 2003, Portland, Oregon.

[4] Batory, D. "A tutorial on feature oriented programming and product-lines." Proceedings of the 25th International Conference on Software Engineering, 2003, pp 753—754.

[5] Bosch, J., "Design of an object-oriented framework for measurement systems," in M. Fayad, D.Schmidt, and R. Johnsson, Eds., Object-oriented application frameworks, Ap 1998.

[6] Bruyninckx, H., "Open robot control software: the OROCOS project," Proc. IEEE International Conference on Robotics and Automation, May 2001, Seoul, Korea.

[7] Clements, P., and Northrop, L., "Software Product Lines: Practices and Patterns," Addison-Wesley Co, Aug. 2001.

[8] Costescu, N.; Loffler, M.; Zergeroblu, E.; Dawson, D., "QRobot—a multitasking PC based robot control system," Proceedings of the 1998 IEEE International Conference on Control Applications. Vol. 2, No. 1-4, Sept. 1998, pp.892—896.

[9] Czarnecki, U. , and Eisenecker, D., "Generative Programming," Addison-Wesley Co. , June 2000.

[10] Elrad, T., Filman, R., and Bader, A., "Aspect-Oriented Programming: Introduction," Communications of the ACM. Vol. 44, No. 10, Oct. 2001, pp. 28-32.

[11] Fowler, M., The New Methodology, Available: http://martinfowler.com/articles/newMethodology.html, April 2003.

[12] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design Patterns," Addison-Wesley Co. , Jan. 1995.

[13] Jung, E., M.S. Thesis, 2005, Dept. of Mechanical Engineering, The University of Texas at Austin.

[14] Kang, K., Cohen, S., Hess, J., Nowak, W., and Peterson, S., "Feature Oriented Domain Analysis (FODA) Feasibility Study," Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, Nov. 1990.

[15] Kapoor, C., and Tesar, D., "Kinematic Abstractions for General Manipulator Control," Proceedings of the 1999 ASME Design Engineering Technical Conferences and Computers in Engineering Conference, Sept. 12-16, 1999, Las Vegas, Nevada.

[16] Kapoor, C. Tesar, D. "A Reusable Operational Software Architecture for Advanced Robotics" Proceedings of the Twelfth CSIM-IFToMM Symposium on theory and Practice of Robots and Manipulators, Paris, France, July 1998.

[17] Loffler, M.S.; Costescu, N.P.; Dawson, D.M., "QMotor 3.0 and the QMotor robotic toolkit: a PC-based control platform," IEEE Control Systems Magazine, Vol. 22, June 2002, pp12-26.

[18] Machine Controller Software Home Page, Robotics Research Group. Available: http://www.robotics.utexas.edu/rrg/research/mcs/.

[19] March, P.; Taylor R.; Kappor, C.; Tesar D., "Decision making for remote robotic operations," Proceedings IEEE Conference on Robotics and Automation 2004, Vol. 3, Apr. 26—May 1, 2004. pp. 2764—2769.

[20] OSCAR Home Page, Robotics Research Group. Available: http://www.robotics.utexas.edu/rrg/research/oscarv.2/

[21] Product Line Architecture Research Group Home Page, University of Texas At Austin. Available: http://www.cs.utexas.edu/users/schwartz/

[22] Pryor, M.; Taylor R.; Kapoor, C.; Tesar, C., "Generalized software components for reconfiguring hyper-redundant manipulators," IEEE/ASME Transactions on Mechatronics. Vol. 7, No. 4, Dec 2002, pp.475—478.

[23] Smaragdakis, Y. and Batory, D., "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs," ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, 2002, pp. 215-255.