

Automated Code Generation for Actuator Interfacing

Ed Jung, Chetan Kapoor

Robotics Research Group, Dept. of Mechanical Engineering
University of Texas at Austin
Austin, TX

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

Abstract - Robot software juggles various coupled concerns, i.e., hardware interfacing, real-time control, computational algorithms, application architecture, etc. Common practice uses object-oriented (OO) frameworks that structure software in terms of objects, classes, and packages. Designers create programs using inheritance and composition. Operational Software Components for Advanced Robotics (OSCAR) is such an object-oriented framework that addresses advanced manipulator control. Though OSCAR applications are developed manually, there is a clear pattern of construction for each class of applications. In spite of these unique, one-off efforts, some common structure for OSCAR applications has emerged. These applications form a *product line*, amenable to automated generation; Product Line Architectures (PLAs), are a theory for such automation. *Feature-Oriented Programming* (FOP) is a PLA method that models and specifies programs in terms of features. Our research applies FOP to robot controller software. As an example, the domain of hardware interfacing is analyzed and 41 features identified. From these 41 features, up to 200 different legal application instances (or configurations) can be created. Our approach using FOP generates these configurations automatically from feature specifications.

Index Terms – Robotics, Product Line, Feature Oriented Programming, Generative Programming, Software Engineering

I. INTRODUCTION

The Robotics Research Group [16] (RRG) has developed the Operational Software Components for Advanced Robotics (OSCAR) framework [12]. 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, as well as OO in general, has shown that OO methods bring improvements in productivity, reusability, and comprehensibility of robot software. Applications, however, are still manually written, one-off efforts, 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 these components. In other words, rapid software assembly requires not just components, but a systematic method for assembling these components.

Feature Oriented Programming (FOP) is one such method for rapid assembly of software [5]. The success of FOP is predicated 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 [18][15]. Many of these control software applications tend to share some common set of base code, as well as design decisions. These programs 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, declaratively.

RRG is applying FOP methods to robot control software, in collaboration with the Product Line Architecture Research Group [17] in Computer Science at UT-Austin. The goals of this paper are to introduce some basic techniques for implementing FOP, and their application to robot software. Specific examples for interfacing with robot actuator hardware are examined.

II. ROBOT SOFTWARE FRAMEWORKS

Object oriented frameworks and design patterns are the dominant methods for software design today. Many such formal and informal frameworks aimed at various levels of robot software have been created. Among more recent efforts are frameworks such as QMotor RTK [13] and Orocos [6]. QMotor is oriented towards servo level and robot level control. Orocos, is recent open source effort divided between two projects -- a generalized library and application framework for robot software, and a hard real-time kernel for all possible feedback control applications. Orocos relies upon design patterns [10] as well as OO methods. The OSCAR framework has been under continuous use and development at RRG for several years [12][16], and counts NASA and DOE [15] amongst its users.

OO frameworks typically provide a set of base classes that represent the infrastructure common to a number of applications. These can be extended through inheritance and polymorphism, or combined through aggregation or composition, to implement specific applications. *Design patterns* recognize that some design problems occur in a common context, and have a common solution [9]. They may be used as a starting point for the design of a new application. This paper considers three examples of OO

TABLE I. SAMPLE OSCAR DOMAINS

Domains	Description
Decision Making	Criteria based decision making software
Device	Abstract interfaces to robots, sensors, actuators, tools, switches, etc.
Dynamics	Lagrangian and Newton-Euler dynamics
Forward Kinematics (FK)	Position, velocity and acceleration level
Inverse Kinematics (IK)	Numerical, closed form and redundant
Motion Planning	Joint space and end effector motion planning and curve criteria
Obstacle Avoidance	Artificial Potential Field based criteria
Performance Criteria	30+ joint and task level criteria and a blackboard architecture for computation

design—from OSCAR, QMotor RTK and a Measurement Systems Framework case study.

A. OSCAR

Though OSCAR is specific to the 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. OSCAR will only be briefly discussed, as more detailed descriptions are provided in later sections.

OSCAR consists of a set of class libraries, a subset of which is described in TABLE I. These classes may be combined or extended in various ways for specific applications. For example, classes in the `InverseKinematics` class library inherit from a base `Kinematics` class and an `IKPosition` class, which provide common interfaces, such as `GetJointPosition` or `SetJointPosition`. Classes such as `IKJacobian`, further extend the inverse kinematics classes for any Jacobian based IK algorithms. `IKPuma` is an extension example for the closed form solution of a Puma robot.

Though these classes are flexible and reusable, there is no *explicit method* or *instruction set* for building applications. In practice, new programs are often created by copy/paste, or based on the past experience and bias of a developer. Clear patterns, however, do exist. These patterns are further enforced by the uniform interface to OSCAR classes, and the data flow inherent to robot control applications. These patterns are different than, but related to the *design patterns* mentioned earlier. Ideally, construction of patterns and applications should be automated.

B. QMotor RTK [13]

The *QMotor RTK (Robotic Toolkit)* is a software package with goals similar 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 is similar to that in OSCAR, in that QMotor makes use of inheritance, polymorphism and OO design methods in its architecture to enable reuse and extensibility. A base class contains common code; extensions implement robot specific code. An example of is shown in TABLE II for a Puma robot, based on QMotor.

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

TABLE II. EXTENSION BY INHERITANCE

```

class PumaControl : public ManipulatorControl
{
    //puma specific code here
    //inherits calculatePositionControl();
};

class PumaPIDControl : public PumaControl {
public:
    void calculatePositionControl() {
        // add PID calculation
        // to Manipulator Control
    }
};

```

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 provided by all robots.

A third option is to factor out the PID code of `PumaPIDControl`, and delegate it to another class. This is essentially the approach of a design pattern, but has its own disadvantages, as will be explained.

C. Measurement Systems Framework [5]

This case study employed two common design patterns for a measurement systems framework—the *Strategy* pattern and the *Factory* pattern [9]. The framework controlled a relatively simple manufacturing system. Sensors measured some property of an object (e.g., weight, size, etc.) on a conveyor. If the value was acceptable, the object passed; if not, an actuator rejected the object. Only the *Strategy* pattern is considered here.

The *Strategy* pattern decouples a class from a specific algorithm by factoring the algorithm into an independent *Strategy* class. The original class may then delegate calls to an instance of the *Strategy* [9]. TABLE III presents an example based on the code of TABLE II.

Sensors used one of several *CalibrationStrategies*, *CalculationStrategies*, and *UpdateStrategies*. 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 client invocation, at a periodic interval, or any time the sensor value changed, respectively.

The authors noted that the *Strategy* pattern allows for a

TABLE III. STRATEGY PATTERN

```

class PumaControl : public ManipulatorControl
{
    //puma specific code here
    void calculatePositionControl() {
        pCaObject->calculatePositionControl();
    }
    ControlAction * pCaObject;
};

class ControlAction {
    virtual void calculatePositionControl()=0;
    ManipulatorControl * pManipulator;
};

class PIDControl : public ControlAction {
public:
    virtual void calculatePositionControl();
};

class FuzzyLogic : public ControlAction {
public:
    virtual void calculatePositionControl();
};

```

“dramatic increase in flexibility,” but also “dramatically complicates” object interactions. Changing or adding a new strategy only requires the instantiation of a new strategy object. The catch is that the *Strategy* object and its parent must then be manually bound to each other.

This complication is evidence of “object schizophrenia” [8]. If a class is broken into smaller fragments, the fragments must be recomposed into a single object. These fragments suffer from “schizophrenia” because they have no reference to the identity of the whole. The case study estimated that up to two thirds of application code simply involved the binding of objects to other objects.

Design patterns tend to fragment an application into many “little classes and methods” [8]. One negative consequence is that the level of granularity becomes much lower, making application construction more difficult.

D. Summary

The main points of the above sections are that (1) OO methods operate at a low level of granularity, (2) OO and pattern methods tend to increase fragmentation of applications, and (3) OO and pattern methods do not explicitly capture high level design decisions or design knowledge. Though OO methods are useful, there is still room for improvement. For large applications, and large scale reuse, the OO methods described do not scale well.

The following steps summarize application design with frameworks: (1) extend existing classes, (2) create new classes, and (3) use 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 in a repeatable and reliable manner. Furthermore, such automation should simplify the process of specifying new programs.

II. FEATURE ORIENTED PROGRAMMING (FOP)

FOP is a technique aimed at the design of *product lines*. 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 [7]. 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 the case for OSCAR applications at the RRG.

In FOP, *features* are the fundamental unit of modularity, rather than classes, objects, or packages. Features represent high level, domain specific abstractions, relevant to a domain expert. A feature encapsulates the software fragments needed to implement the feature. A specific application may then be specified in terms of its features.

These features, which represent fragments of programs, 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 refinements*, where each feature builds upon an existing

Table IV. PUMA MIXIN LAYERS

```

1  /*-----Puma feature-----*/
2  class PumaServos {virtual void setPosition(); };
3  /*-----Standard robot interface feature-----*/
4  template <class parent>
5  class RobotInterface : public parent {
6  public:
7      class ControlAction          {
8      public:
9          void calculatePositionControl()=0;
10         RobotInterface * pRobotInterface;
11     };
12
13     virtual void setPosition() {
14         m_pCaObject->calculatePositionControl();
15     }
16     ControlAction *pCaObject;
17 };
18 /*-----PID control feature-----*/
19 template <class parent>
20 class PID : public parent {
21 public:
22     class ControlAction : public parent::ControlAction {
23         void calculatePositionControl() { /*do pid law*/ }
24         float Kp, Ki, Kd;
25     };
26 };
27 /*-----Fuzzy control feature-----*/
28 template <class parent>
29 class FuzzyLogic : public parent {
30 public:
31     class ControlAction : public parent::ControlAction {
32         void calculatePositionControl() { /*do fuzzy law*/ }
33         float high, medium, low;
34     };
35 };
36 /*-----Program Specifications-----*/
37 typedef RobotInterface <PumaServos> PumaBase; //PumaBase = {RobotInterface, PumaServos}
38 typedef PID< PumaBase > PIDPuma; //PIDPuma = {PID, PumaBase}
39 typedef FuzzyLogic< PumaBase > FuzzyPuma; //FuzzyPuma = {FuzzyLogic, PumaBase}

```

program by adding some code. Thus a complete, existing program may be augmented by a feature, giving it additional capabilities.

The theory and technique for implementing FOP are beyond the scope of this paper, but details on FOP and related program composition technologies are available elsewhere [17][8]; the immediate purpose here is to demonstrate some basic techniques and advantages of using FOP. The fundamental idea is *feature refinement*. There are various methods for implementing refinements. The most straightforward is *mixins* or *mixin layers* [19].

A. Implementing Refinements

Mixins or *mixin layers* are a technique for implementing feature refinement that requires only a C++ compliant compiler with template support. Mixins are also known as *abstract subclasses*. A mixin is a code fragment whose parent class is parameterized. Thus they can be “mixed in” to arbitrary classes. Several mixins can be composed together to create a complete program.

A mixin layer simultaneously refines several classes, in contrast to a mixin, which refines a single class. An example of *mixin layers* on the Puma example of TABLE II is presented in Table IV. This code demonstrates automated assembly of the *Strategy* pattern, with `ControlAction` as the *Strategy*.

Note that a feature refinement may extend classes (lines 5, 19, 21, 30), add classes (line 7), or aggregate objects (lines 10, 16). Feature refinement is thus a technique for automating framework extension [2].

Mixin layers using templates can be difficult to debug for very large systems. Therefore, special tools have been developed by [17] 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 [17]. Understanding of the basic mechanism of mixin layers and feature refinement is all that is immediately necessary.

III. FEATURES IN ROBOT SOFTWARE



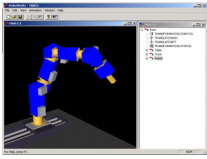
Domain modelling is the process of identifying features in software for a specific domain. Thus it is necessary to identify the features of robot applications.

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.

TABLE V. FEATURES SUPPORTED BY DIFFERENT ROBOT SYSTEMS

KB2017	PowerCube	RoboWorks
		
17 DOF System -P, V, T Control -P Range Limits -P Excess Limits	1-7 DOF System -P, V, C Control -P, V, C Range Limits -P Excess Limits	-P Control
<i>P – Position, V – Velocity, T – Torque, C – Current</i>		

The lower DI layer is used to communicate with different actuators, sensors, tools, etc., as well as to isolate the CC layer from differences among hardware. This paper applies FOP to the DI layer only.

The division of software among these layers is based roughly on timing requirements, so the DI layer requires real time or near real time control, while timing requirements for the CC and HMI layers are less stringent.

A. Device Interface (DI) 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.

The second purpose of the DI layer is to provide real time capabilities. Different types of robot actuators may have different timing requirements, i.e., each robot may require actuator commands to be sent at a different rate. The DI layer should provide actuator commands to the robot’s embedded controller at a consistent rate, and buffer any differences between the production and consumption rates of the CC and DI layers.

These purposes define the two categories of features needed by DI—*Actuator Hardware* and *Real Time*.

B. Actuator Hardware Features

The *Actuator Hardware* features represent the different capabilities of the robots used by RRG. These are the KB2017 dual arm robot with 17 dof, the Powercube modular robot with 1-7 dof, and the Roboworks simulation environment [19], which is a virtual robot interface. These robots and their features are summarized in TABLE V.

Each robot directly supports various features in hardware, such as *actuator command modes* (i.e. Position, Velocity, Current, Torque), *actuator range limits* (i.e. position limits, velocity limits, etc.), *actuator 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 actuator commands outside an acceptable range, while the excess limits features trap *changes* in actuator commands that exceed an acceptable value.

TABLE VI. DI FEATURE SUMMARY

PCoInt CCoInt	VCoInt TCoInt	PRaInt CRaInt	VRaInt TRaInt	PEXInt CEXInt	VEXInt TEXInt
PCoEm	VCoEm	PRaEm CRaEm	VRaEm TRaEm	PEXEm CEXEm	VEXEm TEXEm
PCoHw CCoHw	VCoHw TCoHw	PRaHw CRaHw	VRaHw TRaHw	PEXHw CEXHw	VEXHw TEXHw
<i>P = Position, V = Velocity, C = Current, T = Torque</i> <i>Co = Control, Ra = Range Limits, Ex = Excess limits</i> <i>Int = Interface, Hw = Hardware, Em = Emulated</i>					

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

Some control types may also be emulated, such as position or velocity control¹. For example, the Roboworks environment only accepts position commands. It may be used to model 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 other real time features necessary to emulate velocity, which will be discussed in the next section.

Some manual coding will always be necessary to create an interface to a robot. For the examples above, the KB2017, Powercube, and Roboworks all use different communications protocols and Applications Programming Interfaces (APIs). The goal for the DI layer is to minimize the amount manual code by reusing the common features.

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 VI. DI FEATURE SUMMARY. For example, there is a *Position Control Interface* feature, a *Position Control Hardware* feature, and a *Position Control Emulation* feature.

C. Real Time Features

Earlier mention was made of emulating velocity control. In this case, the DI layer must run at fixed rate to accurately calculate the derivative of position, and several *Real Time* features are then required. These features, listed in TABLE VII, add code for multi-threading and synchronization.

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, in which case DI executes asynchronously, or a single-threaded mode, in which case DI executes synchronously with the CC layer.

The **AvgTuning | FixedTuning** features provide different policies for adjusting the execution rate of a multi-threaded DI program, and the **MotionTime** feature

¹Torque and current control execution are theoretically possible, but would require full inverse dynamic models, which would negatively affect the timing of the DI layer.

TABLE VII. REAL TIME FEATURES

Feature	Description
Locking Nonlocking	Thread-safe or not thread safe code
Active Passive	Multi or single-threaded code
AvgTuning FixedTuning	Different policies for adjusting execution rate
MotionTime	Time step for actuator servo commands
<i>"A B" means choose one of A or B</i>	

simply adds a data member for storing the time step between actuator commands. Detailed explanation is available in [10][14].

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 set {**Locking**, **Active**, **MotionTime**, **AvgTuning**}. If the standard multi-threading mode provided by **Active** is not appropriate for a given application, an application specific version of the **Active** feature can be implemented. Alternatively, a user could simply specify the *Real Time* feature set {**Locking**}, which would make DI thread-safe, and add their concurrency code on top of the DI program.

IV. EXAMPLES

Example 1: Powercube Hardware Interface

A user might want a simple DI program for the Powercube robot. Because the Powercube directly supports several features in hardware, these are automatically chosen. These features are {**PCoHw**, **VCoHw**, **CCoHw**, **PRaHw**, **VRaHw**, **CRaHw**, **PEXHw**}. Next, features which provide an interface into the implementation must be chosen. These features are {**PCoInt**, **VCoInt**, **CCoInt**, **PRaInt**, **VRaInt**, **CRaInt**, **PEXInt**}. No emulated features are desired, so none are chosen.

Because this DI program will be simple, the multi-threading features provided by the *Real Time* features are not required. Therefore the set of *Real Time* features is {**Nonlocking**, **Passive**, **MotionTime**}.

The final specification of features is then {**PCoInt**, **VCoInt**, **CCoInt**, **PRaInt**, **VRaInt**, **CRaInt**, **PEXInt**, **PCoHw**, **VCoHw**, **CCoHw**, **PRaHw**, **VRaHw**, **CRaHw**, **PEXHw**, **Nonlocking**, **Passive**, **MotionTime**}.

Example 2: Powercube Simulation

A user might also want to simulate a Powercube in Roboworks before transferring a program to the actual hardware. Several features then need to be emulated.

Again starting with the Hw features, only **PCoHw** is needed, as Roboworks only supports Position Control. Then the **PCoInt** feature must be chosen, to add an interface to the **PCo** implementation.

Several features must also be emulated. Namely, {**VCoEm**, **PRaEm**, **VRaEm**, **PEXEm**}. Current control (**CCoEm**) cannot be emulated. The corresponding interface features must then be specified, giving {**VCoInt**, **PRaInt**, **VRaInt**, **PEXInt**}.

To emulate velocity control, the DI program must run asynchronously at a steady rate of execution. Therefore, some of the more advanced Real Time features are required. This set of features could either be {Locking, Active, MotionTime, FixedTuning}, or {Locking, Active, MotionTime, AvgTuning}. One final set would be {Locking, Active, MotionTime, AvgTuning, VCoInt, PRaEm, VRaEm, PEXEm, PRaInt, VRaInt, PEXInt, VCoEm}.

V. CONCLUSION

FOP is a technique for designing software product lines. In contrast to OO techniques, FOP models programs in terms of features, which operate at a much higher level of abstraction than classes or objects. Programs may be specified using features, 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* and *Real Time* features, was used to model this software. Examples for specifying two different program variants were also presented.

There are many possible scenarios other than the two examples presented. Because the DI domain is relatively simple, it is possible to exhaustively enumerate the total number of valid configurations, by following a process similar to the examples above for each robot.

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, depending upon the specific robot. A total of 200 different configurations for the entire model are summarized in TABLE VIII.

For large scale systems, the explosion in the number of feature combinations can result in multiple feature compositions that suit application requirements. At this point, optimization of the specification based on secondary objectives is required. Example objectives in the robotics domain are, computational resource constraints, memory footprint, bandwidth and number of IO channels, etc.

One issue not addressed here is the *validation* of a program specification. A user might potentially specify an incorrect combination of features, or an incorrect ordering of features. These two problems are addressed by the definition of a *grammar* and *design rules* [1], which would automatically enforce design constraints. Additionally, it is possible to exploit the structure of the *Actuator Hardware* features, as is evident in TABLE VI, to simplify the specification [3]. This is, in fact, how the

configuration counts were generated.

Another aspect is related to formal verification of the software. Ideally, this should be performed at the feature level, with guarantees on validation of the application instance. At this point, our approach is to generate test programs for each feature. As features are composed, these test programs are also composed, resulting in a program that tests the overall composition.

ACKNOWLEDGMENTS

We would like to thank Prof. D. Batory of [17] for providing tool support and advice on this work.

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. Johnson, 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] Czarnecki, U. , and Eisenecker, D., "Generative Programming," Addison-Wesley Co. , June 2000.
- [9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design Patterns," Addison-Wesley Co. , Jan. 1995.
- [10] Jung, E., M.S. Thesis, 2004, The University of Texas at Austin.
- [11] 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.
- [12] Kapoor, C. Tesar, D. "A Reusable Operational Software Architecture for Advanced Robotics" Proceedings of the Twelfth CSIM-IFTToMM Symposium on theory and Practice of Robots and Manipulators, Paris, France, July 1998.
- [13] 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.
- [14] Machine Controller Software Home Page, Robotics Research Group: <http://www.robotics.utexas.edu/rrg/research/mcs/>.
- [15] March, P.; Taylor R.; Kappor, C.; Tesar D., "Decision making for remote robotic operations," Proc. IEEE Conf. on Robotics and Automation 2004, Vol. 3, Apr. 26-May 1, 2004. pp. 2764—2769.
- [16] OSCAR Home Page, Robotics Research Group. Available: <http://www.robotics.utexas.edu/rrg/research/oscarv.2/>
- [17] Product Line Architecture Research Group Home Page Available: <http://www.cs.utexas.edu/users/schwartz/>
- [18] 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.
- [19] Roboworks Home Page, Available: <http://www.newtonium.com>.
- [20] 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.

TABLE VIII. CONFIGURATION COUNT

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